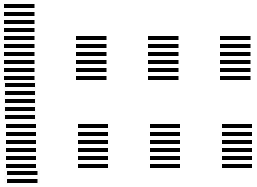
Introduction to OpenMP

(https://computing.llnl.gov/tutorials/openMP/)

- An Application Program Interface (API) which provides a portable and scalable model for developers of shared memory parallel applications (so that they do not have to deal with pthread)
  - A library with some simple functions
  - A few program directives (pragmas = hints to the compiler)
  - A few environment variables

- A compiler translates OpenMP functions and directives to Pthread calls.
- Program begins with a Master thread
- Threads are forked when specified by directives



Specifying a parallel region

```
#include <omp.h>
int main(){
print(“The output:\n”);
#pragma omp parallel /* define multi-thread section */
 {
   printf(“Hello World\n”);
 }
/* Resume Serial section*/
 printf(“Done\n”);
}
```

```
The output:
Hello World
Hello World
Hello World
Done
```

- The number of forked threads can be specified through:
  - o An environment variable: `setenv OMP_NUM_THREADS 8`
  - o An API function: `void omp_set_num_threads(int number);`

- Can also get the number of threads by calling
  `int omp_get_num_threads();`

Threads id and private variables

```
#include <omp.h>
int main()  {
  int id, np;
#pragma omp parallel private(id, np)
  {
    np = omp_get_num_threads();
    id = omp_get_thread_num();
    printf(“Hello from thread %d out of %d threads\n”, id, np);
  }
}
```

```
Hello from thread 0 out of 3
Hello from thread 1 out of 3
Hello from thread 2 our of 3
```

- The format of a pragma in C/C++ is:

  `#pragma omp name_of_directive [optional_clauses]`

- “private” is a “clause”. It declares variables that are local to the forked threads.
- A similar clause, “shared” is used to declare variables that are not local (the default).

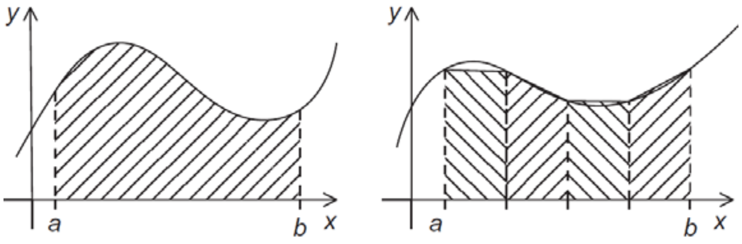
The num_threads clause

- The num_threads clause can be added to a parallel directive.
- It allows the programmer to specify the number of threads that should execute the following block.

pragma omp parallel num_threads (thread_count)

- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn’t guarantee that this will actually start thread_count threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we’re trying to start a lot of threads, we will almost always get the desired number of threads.

Example: the trapezoidal rule



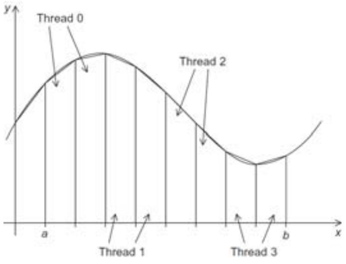
```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```

A First OpenMP Version

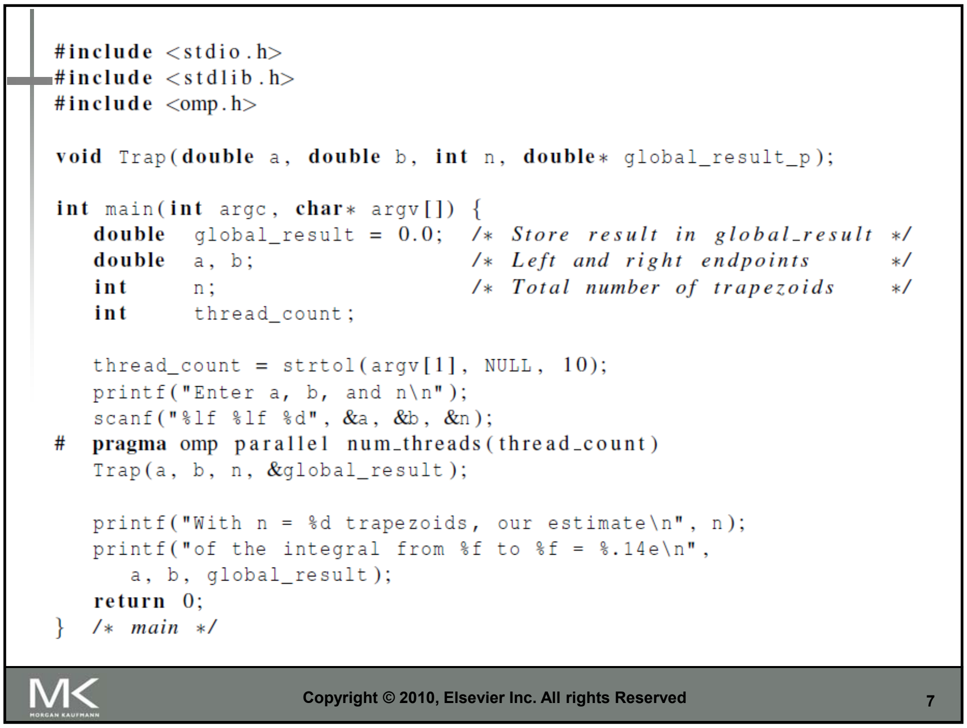
1) We identified two types of tasks:
   a) computation of the areas of individual trapezoids,
   b) adding the areas of trapezoids.

2) There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1b.

3) We assumed that there would be many more trapezoids than cores.



- So we aggregated tasks by assigning a contiguous block of trapezoids to each thread/core.



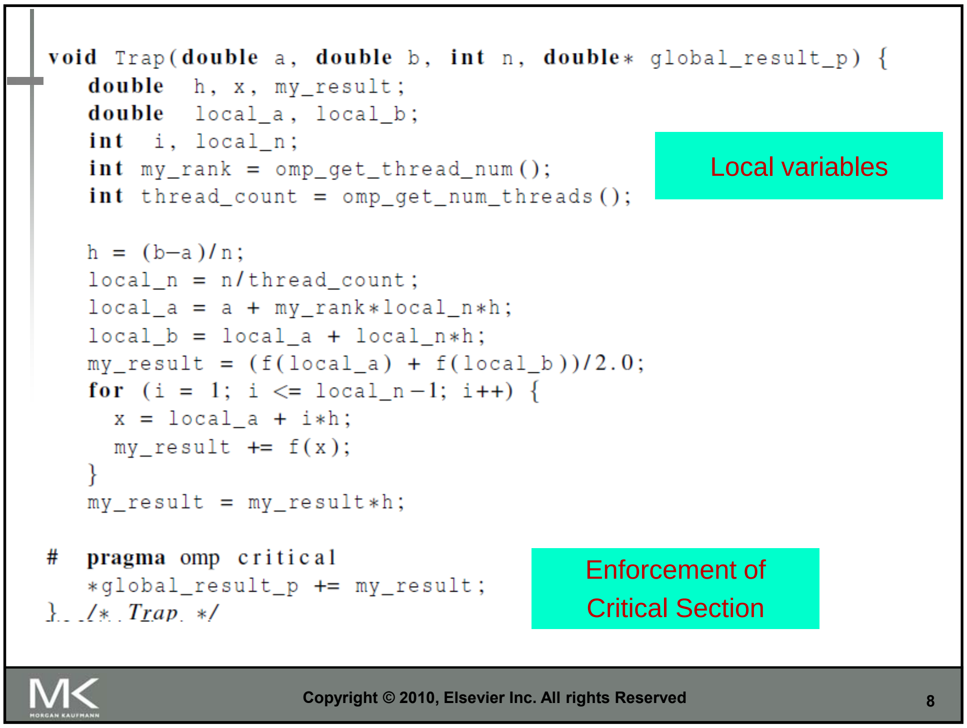
```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```



```
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

Local variables

Enforcement of Critical Section

Parallel For loops and the For pragmas

```
#define N 1000
#define N_THREADS 4
main ()  {
  int chunk, a[N], b[N], c[N];
  omp_set_num_threads(N_THREADS);
  chunk = N / N_THREADS ;
 . . .
#pragma omp parallel shared(a,b,c) private(i, id)
  {
    id = omp_get_thread_num();
    for (i=id * chunk; i < (id+1) * chunk; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

OMP can automatically distribute the work to the threads

```
#pragma omp for
  { for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
  }
```

Combining the Parallel and For pragmas (work-sharing)

```
. . .
#pragma omp parallel shared(a,b,c) private(i)
{
  #pragma omp for
  { for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
  }
}
```

```
#pragma omp parallel for shared(a,b,c) private(i)
  { for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
  }
```

NOTE: it is the responsibility of the programmer to make sure that there is no loop dependencies

Legal forms for parallelizable for statements

```
                                          index++
                                          ++index
                        index < end       index--
                        index <= end      --index
for ( index = start  ;  index >= end  ;   index += incr
                        index > end       index -= incr
                                          index = index + incr
                                          index = incr + index
                                          index = index - incr   )
```

- The variable index must have integer or pointer type (e.g., can't be float).

- The expressions start, end, and incr must have a type compatible with index and must not change during execution of the loop. The variable index can only be modified by the "increment expression" in the for statement.

Data dependencies

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < 10; i++)
   fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

↓

note 2 threads

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < 10; i++)
   fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

May get the correct output: 1 1 2 3 5 8 13 21 34 55

But may also get: 1 1 2 3 5 8 0 0 0 0

Can you guess why?

Other clauses for data scoping

Shared (…)
Private(…)
Firstprivate(…) /* private but initialized to its value before entering the region*/
Lastprivate (…) /* private but on exit, value in master = last value in thread*/

Defaults(Private | shared | none) /* default for all variable in the thread */

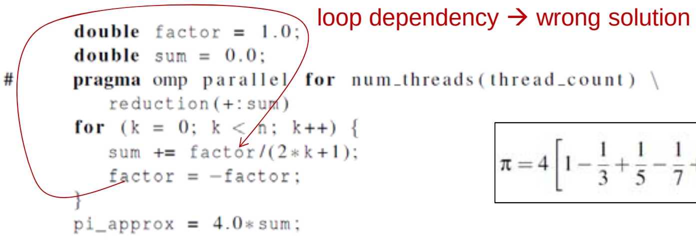
Reduction (operator: list) /* variables declared in enclosing context, are private in the parallel sections, but reduced upon exit using the specified operator (e.g. +, *, -, &, |, ^, &&, ||) */

```
main () {
int i, n;
float a[100], b[100], result;
n = 100; result = 0.0;
for (i=0; i < n; i++) { a[i] = … ; b[i] = … ; }

#pragma omp parallel for default(shared) private(i) reduction(+:result)
   for (i=0; i < n; i++)
       result = result + (a[i] * b[i]);

printf("Final result= %f\n",result); }
```

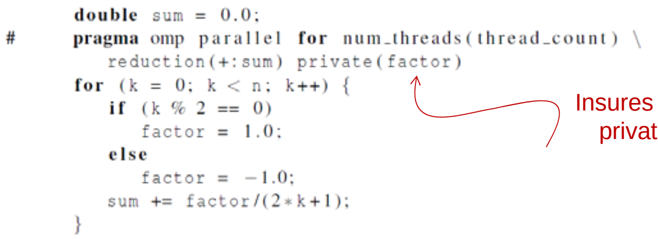
OpenMP for estimating π



loop dependency → wrong solution

```
    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;
```

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$



```
    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);
    }
```

Insures factor has private scope.

The “sections” work-sharing directive

```
#define N 1000
main (){
 int i;float a[N], b[N], c[N];
 for (i=0; i < N; i++) a[i] = b[i] = … ;

#pragma omp parallel shared(a,b,c) private(i)
 {
  . . .
 #pragma omp sections
  {
  #pragma omp section
   {
   for (i=0; i < N/2; i++)
    c[i] = a[i] + b[i];
   }
  #pragma omp section
   {
   for (i=N/2; i < N; i++)
    c[i] = a[i] + b[i];
   }
  }  /* end of sections */
 . . .
}  /* end of parallel section */
}
```

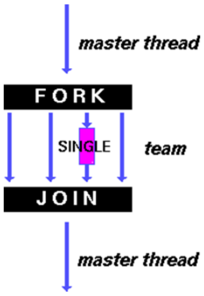
Different code in each section.

- **May combine the parallel and sections pragmas (as we did with for)**
- **If more threads than sections, then some are idle**
- **If fewer threads than sections, then some sections are serialized**

The “Single” and “Master” directive

- The “single” directive serializes a section of code within a parallel region
- More convenient and efficient than terminating a parallel region and starting it later
- Typically used to serialize a small section of the code that’s not thread safe
- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a “nowait” clause is specified

- The “Master” directive is the same as the “single” directive except that:
  - The serial code is executed by the *master* thread
  - The other threads skip the master section, but do not wait for the *master* thread to finish executing it.



The Critical and barrier directives

```
#pragma omp parallel for shared(sum)
  for(i = 0; i < n; i++){
    value = f(a[i]);
    #pragma omp critical(name)
    {
      sum = sum + value;
    }
  }
```

- A critical section, executed by one thread at a time.
- (name) is optional
- Critical sections with different names can be executed simultaneously

```
#pragma omp barrier
```

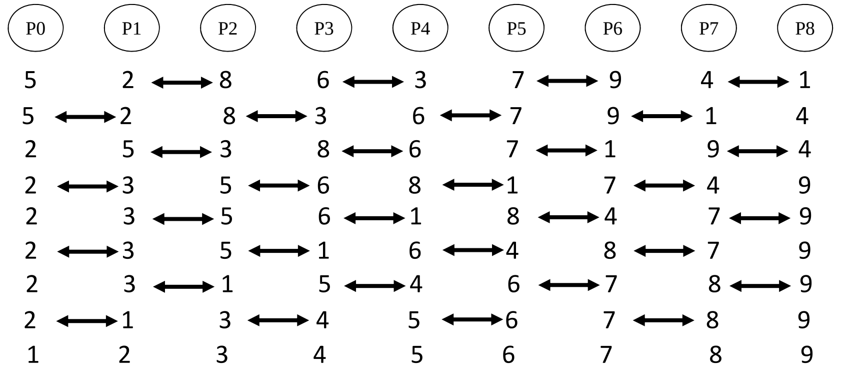
Exactly what you would expect from a barrier

The Atomic directive

```
#pragma omp parallel for shared(sum)
  for(i = 0; i < n; i++){
    value = f(a[i]);
    #pragma omp atomic
     sum = sum + value;
  }
```

- A critical section composed of one statement (expression) of the form, x++ , x--, ++x, --x or x <op> = <expression>;
- <op> can be +, *, -, /, &, ^, |, << or >>
- Compiler will use hardware atomic instructions
- Is more efficient than using the critical section directive

Example Odd-Even Transposition Sort



P0		P1		P2		P3		P4		P5		P6		P7		P8
5		2	↔	8		6	↔	3		7	↔	9		4	↔	1
5	↔	2		8	↔	3		6	↔	7		9	↔	1		4
2		5	↔	3		8	↔	6		7	↔	1		9	↔	4
2	↔	3		5	↔	6		8	↔	1		7	↔	4		9
2		3	↔	5		6	↔	1		8	↔	4		7	↔	9
2	↔	3		5	↔	1		6	↔	4		8	↔	7		9
2		3	↔	1		5	↔	4		6	↔	7		8	↔	9
2	↔	1		3	↔	4		5	↔	6		7	↔	8		9
1		2		3		4		5		6		7		8		9

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

How many phases are enough?

First OpenMP Odd-Even Sort

```
      for (phase = 0; phase < n; phase++) {
         if (phase % 2 == 0)
#           pragma omp parallel for num_threads(thread_count) \
               default(none) shared(a, n) private(i, tmp)
            for (i = 1; i < n; i += 2) {
               if (a[i-1] > a[i]) {
                  tmp = a[i-1];
                  a[i-1] = a[i];
                  a[i] = tmp;
               }
            }
         else
#           pragma omp parallel for num_threads(thread_count) \
               default(none) shared(a, n) private(i, tmp)
            for (i = 1; i < n-1; i += 2) {
               if (a[i] > a[i+1]) {
                  tmp = a[i+1];
                  a[i+1] = a[i];
                  a[i] = tmp;
               }
            }
      }
```

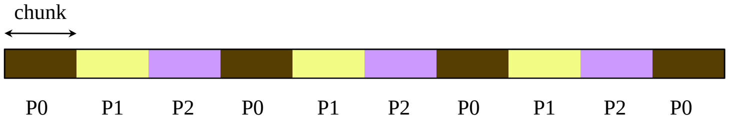
Second OpenMP Odd-Even Sort

```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Is this version more or less efficient than the first one?

Loop Scheduling in OpenMP

- **Static scheduling (the default scheduling):**
  - Iterations space is divided into chunks and assigned to threads statically in a round robin fashion.

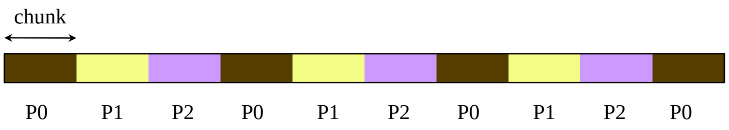


Examples for scheduling *i* =0,1,2, … , 23 on 4 threads:

	Thread 1	Thread 2	Thread 3	Thread 4
Chunk = 1	0,4,8,12,16,20	1,5,9,13,17,21	2,6,10,14,18,22	3,7,11,15,19,23
Chunk = 2	0,1,8,9,16,17	2,3,10,11,18,19	4,5,12,13,20,21	6,7,14,15,22,23
Chunk = 3	0,1,2,12,13,14	3,4,5,15,16,17	6,7,8,,18,19,20	9,10,11,21,22,23
Chunk = 4	0,1,2,3,16,17,18,19	4,5,6,7,20,21,22,23	8,9,10,11	12,13,14,15

Loop Scheduling in OpenMP

- **Dynamic scheduling**:
  - Iterations are divided into chunks and assigned to threads dynamically.
  - Each thread executes a chunk, and when done, requests another chunk from the run-time system.
  - Although each chunk contains the same number of iterations, chunks may have different execution times.



Is this better than static?
what is the effect of the chunk size?

Example

We want to parallelize:

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

Note: f(i) calls sin() *i* times.

Where f() is defined by:

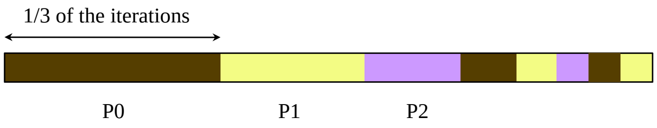
```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}  /* f */
```

- Results when n=10,000:
  - default scheduling→ run-time = 2.76 seconds
  - Cyclic static scheduling → run-time = 1.84 seconds
  - Do you think dynamic scheduling will do better?

Why is that?

Loop Scheduling in OpenMP



- **Guided scheduling:**
  - At the dynamic scheduling decision, the chunk size = 1/P of the remaining iterations, where P is the number of threads.
  - Can specify the smallest chunksize (except possibly the last).
  - The default smallest chunksize is 1

The scheduling scheme and the chunk size is determined by a “schedule” clause in the **opm for** directive. For example:

```
int chunk = 3
#pragma omp parallel for shared(a,b,c,chunk) \
       private(i) schedule(guided,chunksize)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];}
```

`or dynamic or static`

The “schedule” clause (optional)

```
#pragma omp parallel for schedule(type,chunksize)
```

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time based on the value of the environment variable OMP_SCHEDULE (can be static, dynamic or guided).
- The chunksize is a positive integer.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

Clauses/Directives summary

Clause	**PARALLEL**	**For**	**SECTIONS**	**SINGLE**	**PARALLEL For**	**PARALLEL SECTIONS**
IF	X				X	X
PRIVATE	X	X	X	X	X	X
SHARED	X	X			X	X
DEFAULT	X				X	X
FIRSTPRIVATE	X	X	X	X	X	X
LASTPRIVATE		X	X		X	X
REDUCTION	X	X	X		X	X
SCHEDULE		X			X	
ORDERED		X			X	
NOWAIT		X	X	X		

- IF clause must evaluate to a non-zero integer for the parallel threads to fork
- ORDERED forces the loop to proceed in serial order (== critical section)
- NOWAIT overrides the barrier implicit in a directive.

Run-time library routines

```
void omp_set_num_threads(int num_threads)
void omp_set_dynamic(int condition) /*enables dynamic adjustment of number
                                      of threads (if condition != 0)*/
int omp_get_num_threads()
int omp_get_thread_num()
int omp_get_num_procs()
int omp_in_parallel()
void omp_init_lock( omp_lock_t *lock )
void omp_set_lock( omp_lock_t *lock )
void omp_unset_lock( omp_lock_t *lock )
void omp_test_lock( omp_lock_t *lock )
void omp_destroy_lock( omp_lock_t *lock )
```

Message passing using queues

- Queues can be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket.
- A natural data structure to use in many multithreaded applications.
- For example, suppose we have several “producer” threads and several “consumer” threads.
  - Producer threads might “produce” requests for data.
  - Consumer threads might “consume” the request by finding or generating the requested data.
- Each thread could have a shared message queue, and when one thread wants to “send a message” to another thread, it could enqueue the message in the destination thread’s queue.
- A thread could receive a message by dequeuing the message at the head of its message queue.

Startup

- When the program begins execution, a single thread, the master thread, will get command line arguments and allocate an array of message queues: one for each thread.
- This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.

- One or more threads may finish writing to their queues before some other threads.
- If we want to synchronize, we need an explicit barrier so that when a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier.
- After all the threads have reached the barrier all the threads in the team can proceed.

Message Passing

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
   Send_msg();
   Try_receive();
}

while (!Done())
   Try_receive();
```

```
   mesg = random();
   dest = random() % thread_count;
#  pragma omp critical
   Enqueue(queue, dest, my_rank, mesg);
```

Sending a message
(put in receiver's queue)

Message Passing

```
   if (queue_size == 0) return;
   else if (queue_size == 1)
#     pragma omp critical
      Dequeue(queue, &src, &mesg);
   else
      Dequeue(queue, &src, &mesg);
   Print_message(src, mesg);
```

Receiving a message (from local queue)

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
   return TRUE;
else
   return FALSE;
```

Termination detection

each thread increments this after completing its for loop