Parallel Programming with OpenMP

OpenMP (Open Multi-Processing)

A directive-based API that can be used with C and C++ for programming *shared memory systems*.

OpenMP is an explicit (not implicit, e.g., Matlab) programming model, offering the programmer full control of parallelization.

- OpenMP directives provide support for concurrency, synchronization, and data handling
- Avoid the need for explicitly setting up mutexes, condition variables, data scope, and initialization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

Ease of use

- Provide capability to incrementally parallelize a serial program

OpenMP Directives

OpenMP code always begins with `# pragma omp`

A directive consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```

OpenMP programs execute serially until they encounter the "parallel" directive, which creates a group of threads.

```
#pragma omp parallel [clause list]
/* structured block */
```

The main thread that encounters the parallel directive becomes the master of this group of threads and is assigned the thread id 0 within the group.

OpenMP: Hello, World

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

void Hello(void);  /* Thread function */

/*--------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */
```

Directives

```
/*--------------------------------------------------------------------
 * Function:    Hello
 * Purpose:     Thread function that prints message
 */
void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Functions

The Parallel Region Construct: “Parallel”

`parallel` **directive: the structured block of code that follows should be executed by multiple threads.**

The default num of threads used is determined by the system

`num_threads` **clause specifies the number of threads:**

```
  # pragma omp parallel num_threads(thread_count)
```

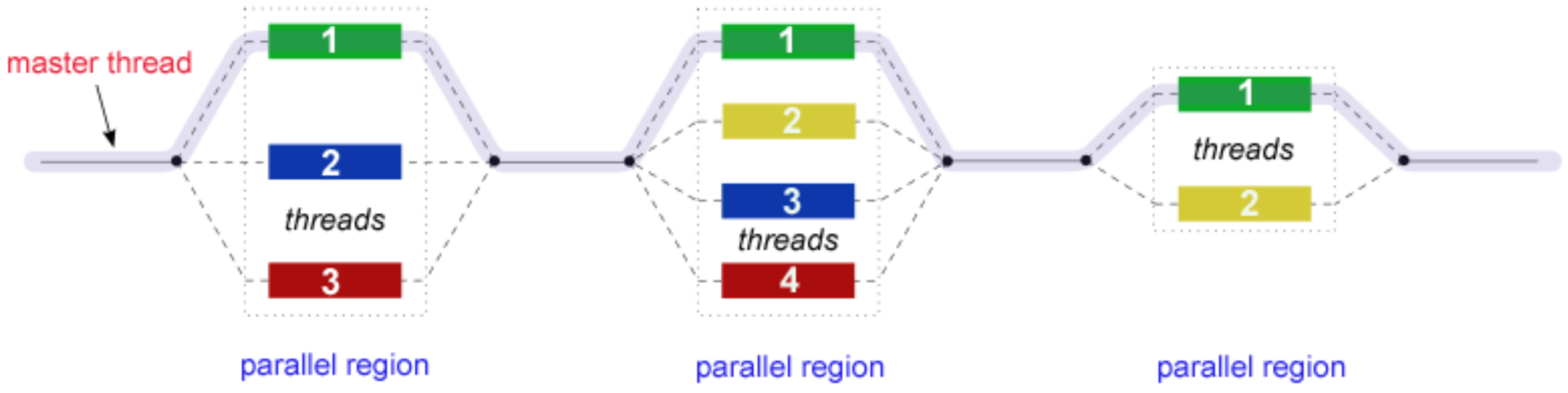
Master: the original thread that calls the parallel directive

Slaves: the started threads

Team: *both* the master *and* slaves

Every thread in the team will execute the block that follows.

Workflow for the Parallel Region Construct



OpenMP uses the fork-join model of parallel execution

FORK: the master thread creates a team of parallel threads.

The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.

JOIN: When the team threads complete the statements in the parallel region construct, they *synchronize* and terminate, leaving only the master thread.

Parallel Region Construct

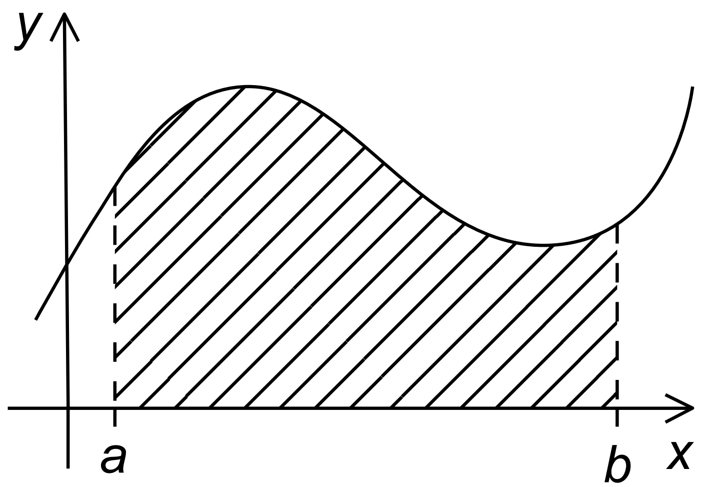
```
#pragma omp parallel [clause ...]
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)
   structured_block
```

An Example: the Trapezoidal Rule

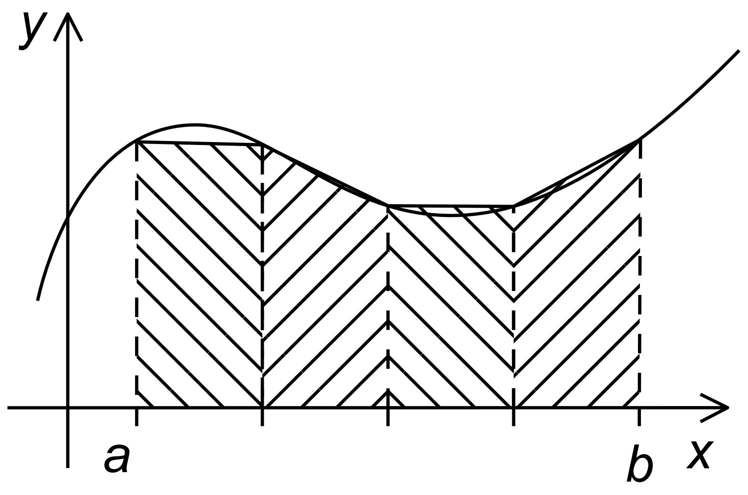
The Trapezoidal rule can be used to estimate the area under a function $y = f(x)$

$$h = (b-a)/n \qquad x_i = a + ih,\ i = 0, 1, \ldots, n,$$

$$\text{Estimated area} = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$



(a)



(b)

Default Variable Scopes

Shared scope: can be accessed by all the threads in the team
Private scope: can only be accessed by a single thread.

By Default:

Variables declared before a parallel directive have shared scope among the threads in the team.

- The value of a shared variable at the beginning of the parallel block is the same as the value before the block
- After completion of the parallel block, the value of the variable is the value at the end of the block.

Variables declared in the block (e.g., local variables in functions) have private scope.

Critical Sections

```
/* In omp_trap1.c */

#  pragma omp critical
   *global_result_p += my_result;



/* In omp_trap2a.c */

   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;

      my_result += Local_trap(a, b, n);
#     pragma omp critical
      global_result += my_result;
   }
```

Reduction Clause (another Data Scope)

Reduction operator: a binary operation (such as addition or multiplication)

Reduction: a computation that repeatedly applies the same reduction operator to a sequence of operands to get a single result.

All intermediate results should be stored in the same reduction variable.

```
/* In omp_trap2b.c, the following does the same as omp_trap2a.c*/

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

OpenMP creates a private variable "my_result" for each thread, and "my_result" are added to the shared "global_result" in a critical section.

The private variables created for a reduction clause are initialized to the identity value for the operator, e.g., 0 for +, 1 for ×.

Work-Sharing Constructs: “for”

Three work-sharing constructs: “for”, “sections”, “single”

A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.

Work-sharing constructs do NOT launch new threads: must be enclosed in a parallel region. The team must be already there!

There is no barrier upon entry to a work-sharing construct; however, there is an implicit barrier at the end of a work sharing construct.

“for” must be preceding a for loop

Default partition of iterations among threads depends on the system

Usually use a block partitioning: each thread handles roughly n/thread_count iterations.

The default scope of the loop variable is *private*; each thread in the team has its own copy of *i*.

Work-Sharing Constructs: “for”

```
/* omp_trap3.c */

   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: approx)
   {
     #  pragma omp for
     for (i = 1; i <= n-1; i++)
       approx += f(a + i*h);
   }
   approx = h*approx;

/* omp_trap3.c (merged version) */

   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
     approx += f(a + i*h);
   approx = h*approx;
```

Work-Sharing Constructs: “for”

```
#pragma omp for [clause ...]
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait
   for_loop
```

Work-Sharing Constructs: "for"

OpenMP will only parallelize for loops for which the number of iterations can be determined

from the for statement (i.e., the code `for (…;…;…)`) itself

AND prior to execution of the loop

Cannot parallelize the following

```
for ( ; ; ) {
    . . .
}
```

```
for (i = 0; i < n; i++) {
   if ( . . . ) break;
   . . .
}
```

Work-Sharing Constructs: “for”

Index must be an integer or a pointer (can’t be `float`)

Index can only be modified by the increment in the for statement

All variables must have compatible types, e.g.,

$$\texttt{for}\left(\texttt{index = start}\quad;\quad\begin{array}{l}\texttt{index < end}\\ \texttt{index <= end}\\ \texttt{index >= end}\\ \texttt{index > end}\end{array}\quad;\quad\begin{array}{l}\texttt{index++}\\ \texttt{++index}\\ \texttt{index--}\\ \texttt{--index}\\ \texttt{index += incr}\\ \texttt{index -= incr}\\ \texttt{index = index + incr}\\ \texttt{index = incr + index}\\ \texttt{index = index - incr}\end{array}\right)$$

Watch out for Loop-Carried Dependences

This is OK, since dependences exist within each iteration

```
#  pragma omp parallel for num_threads(thread_count)
   for (i = 0; i < n; i++) {
      x[i] = a + i*h;
      y[i] = exp(x[i]);
   }
```

The following is Wrong, although can be compiled! There are dependences between iterations (threads), which OpenMP doesn't detect for you.

```
   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];
```

Frequently Used Data Scope Clauses

`private(list of variables)`

private to each thread (initialized to random value)

`shared(list of variables)`

shared among the team

`default(none | shared | private)`

if `none`, specify the data scopes explicitly using other clauses.

`firstprivate(list of variables)`

private, and initialized to the value of original objects before entry into the parallel region.

`lastprivate(list of variables)`

private, with the value obtained from the last (sequential) iteration or section, copied back into the original variable object.

`reduction(operator: list of variables)`

private vs. firstprivate

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

int main (void)
{
    int i = 10;

    #pragma omp parallel private(i)
    {
        printf("thread %d: i = %d\n", omp_get_thread_num(), i);
        i = 1000 + omp_get_thread_num();
    }

    printf("i = %d\n", i);

    return 0;
}
```

private vs. first private

With four threads, the output is

```
thread 0: i = 0
thread 3: i = 32717
thread 1: i = 32717
thread 2: i = 1
i = 10
```

```
(another run of the same program)
```

```
thread 2: i = 1
thread 1: i = 1
thread 0: i = 0
thread 3: i = 32657
i = 10
```

If i is made `firstprivate`, then the output would be

```
thread 2: i = 10
thread 0: i = 10
thread 3: i = 10
thread 1: i = 10
i = 10
```

Example: Computing PI

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

```
        double 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;
```

Is this correct? Why or why not?

Example: Computing PI

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

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           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);
        }
```

The correct version to compute $\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$

Example: Bubble Sort

Can we parallelize this using OpenMP?

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

Example: Odd-Even Transposition Sort

Can we parallelize this with OpenMP?

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

	Subscript in Array						
Phase	**0**		**1**		**2**		**3**
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

Example: Odd-Even Transposition Sort

OpenMP Version 1

```
   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;
            }
         }
   }
```

Example: Odd-Even Transposition Sort

OpenMP Version 2

```
#  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;
            }
         }
   }
```

Example: Odd-Even Transposition Sort

Remember there is an implicit barrier at the end of “for”

One fork and join with “parallel for” per iteration

Run time comparison of “parallel for” and “for”

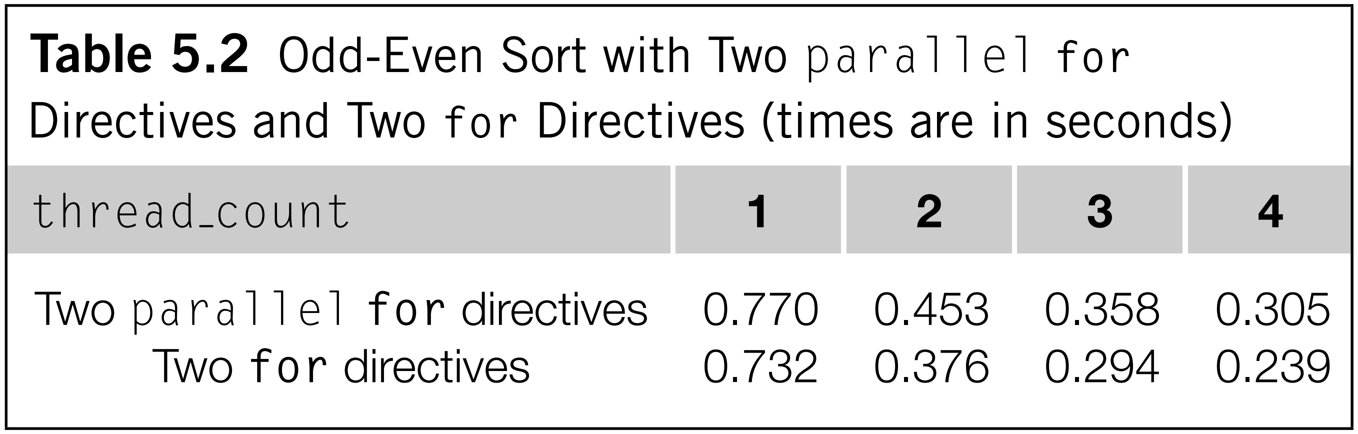


Table 5.2 Odd-Even Sort with Two `parallel for` Directives and Two `for` Directives (times are in seconds)

`thread_count`	**1**	**2**	**3**	**4**
Two `parallel for` directives	0.770	0.453	0.358	0.305
Two `for` directives	0.732	0.376	0.294	0.239

Scheduling For Loops

`schedule (type [,chunksize])`

STATIC type

Iterations assigned before loop execution

Assigns chunks of `chunksize` iterations to each thread in a round-robin fashion.

By default, `chunksize` is approximately

number_of_iterations / number_of_threads

DYNAMIC type

Iterations assigned dynamically during loop execution

Loop iterations are divided into chunks of size `chunksize`. Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system. The default `chunksize` is 1.

Scheduling For Loops

For example, with 3 threads 9 iterations, we have the following assignments

`schedule (static):`

tid	List of iterations
0	0, 1, 2
1	3, 4, 5
2	6, 7, 8

`schedule (static,1):`

tid	List of iterations
0	0, 3, 6
1	1, 4, 7
2	2, 5, 8

`schedule (dynamic):`

Depends on the execution order of threads

Scheduling For Loops

GUIDED type

Iterations assigned dynamically during loop execution

Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.

Similar to dynamic type except that `chunksize` decreases each time a chunk of work is given to a thread. The size of the initial chunk is proportional to:

> *number_of_iterations / number_of_threads*

Subsequent chunks are proportional to

> *number_of_iterations_remaining / number_of_threads*

The `chunksize` defines the minimum chunk size. The default `chunksize` is 1.

> In a guided schedule, it decreases down to `chunksize`, with the exception that the last chunk can be smaller.

Scheduling For Loops

GUIDED type example

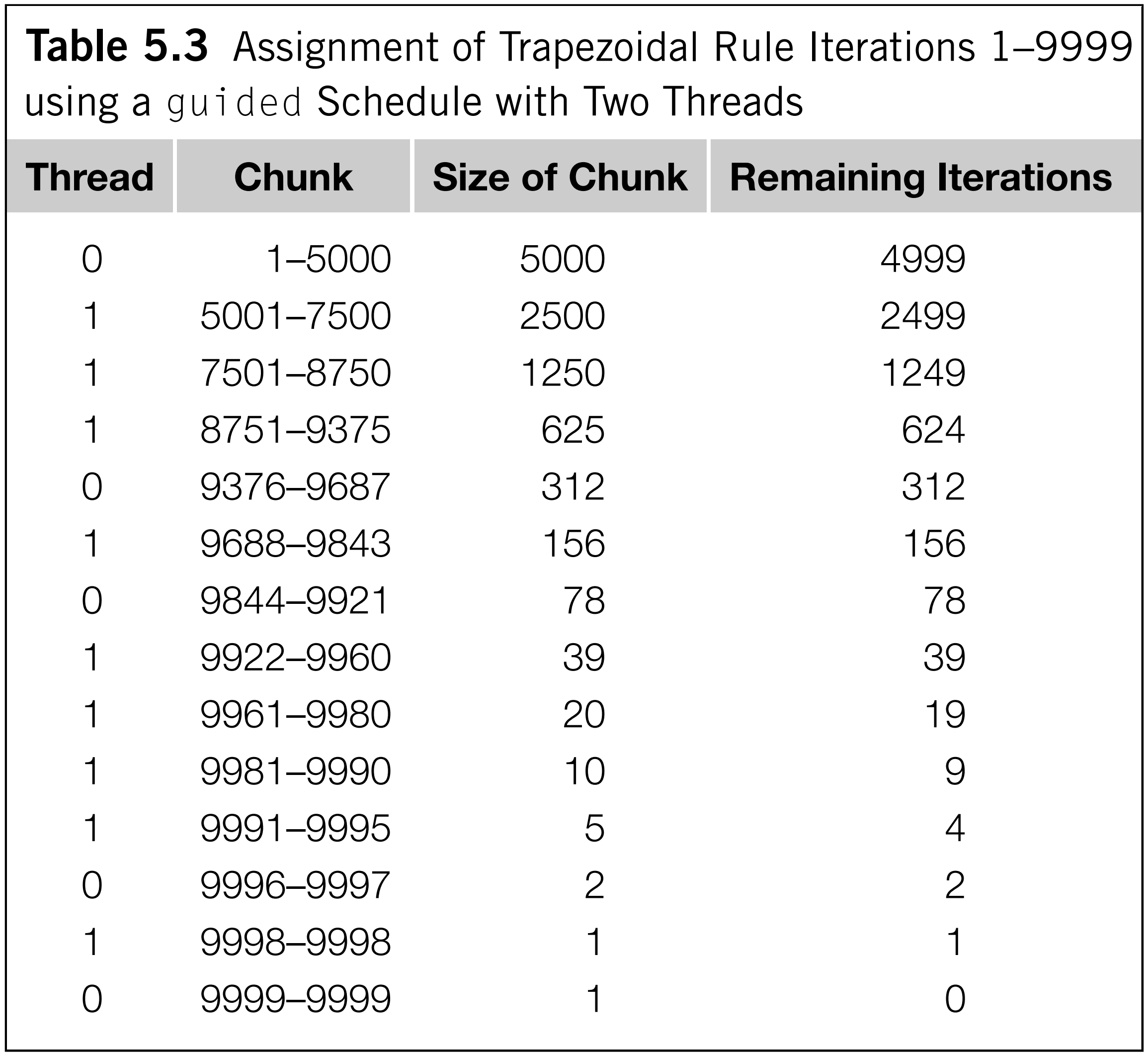


Table 5.3 Assignment of Trapezoidal Rule Iterations 1–9999 using a `guided` Schedule with Two Threads

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

Scheduling For Loops

RUNTIME type

The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

```
export OMP_SCHEDULE=“static,1"
```

AUTO type

The scheduling decision is delegated to the compiler and/or runtime system.

DEFAULT type

It is implementation dependent.

Other Clauses in the For Construct

nowait

If specified, then threads do NOT synchronize at the end of the parallel loop.

ORDERED

Specifies that the iterations of the loop must be executed as they would be in a serial program.

COLLAPSE

Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause.

The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

OpenMP Ordered

```
#pragma omp parallel for ordered schedule(dynamic,3)
    for (int i = 0; i < n; ++i){
            ...
            ...
            ...
#pragma omp ordered
            v.push_back(i);
    }
```

Different threads execute concurrently until they encounter the ordered region, which is then executed sequentially in the same order as it would get executed in a serial loop.

Nested Loops and Collapse

```
#pragma omp parallel for collapse(2)
  for (i = 0; i < imax; i++) {
     for (j = 0; j < jmax; j++)
        a[j + jmax*i] = 1;

  }
```

Use the collapse-clause to increase the total number of iterations that will be partitioned

Reduce the granularity of work to be done by each thread.

To use "collapse()"

- Loop needs to be perfectly nested
- Loop needs to have rectangular iteration space
- Makes iteration space larger
- Less synchronization needed than nested parallel loops

Sections Construct

Sections/Section:

Assigns the structured block corresponding to each section to one thread (more than one section can be assigned to each thread).

Like "for" construct, there is an implicit barrier after it.

```
#pragma omp sections [clause ...]
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
{
   #pragma omp section
      structured_block
  #pragma omp section
      structured_block
}
```

Sections Construct

```
#include <omp.h>
#define N 1000

main (){

int i;
float a[N], b[N], c[N], d[N];

for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
}

#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */
  }  /* end of parallel section */
}
```

Single Construct

Single:

Enclosed block is to be executed by only one thread in the team

There is an **implicit barrier** at the end of the block (`nowait` clause can cancel it)

Examples: performing I/O, computing a global variable, e.g., mean

```
void work1() {}
void work2() {}
void a12() {

#pragma omp parallel {

   #pragma omp single
   printf("Beginning work1.\n");

   work1();

   #pragma omp single
   printf("Finishing work1.\n");

   #pragma omp single nowait
   printf("Finished work1 and beginning work2.\n");

   work2();
}
```

OpenMP Synchronization Constructs

Critical: `#pragma omp critical`

Restricts the access to a critical section by one thread a time

Master: `#pragma omp master`

Enclosed block will be executed by only the master thread in the team

There is NO implicit barrier after it

Barrier: `#pragma omp barrier`

Enforce all threads to wait at an explicit barrier

Example: Find the Max from an Array

```
/* First Edition */

int largest = 0;
#pragma omp parallel for
for ( int i = 0; i < 1000; i++ ) {
  #pragma omp critical
  if (data[i] > largest)
     largest = data[i];
}

/* Second Edition */

int largest = 0;
#pragma omp parallel for
for ( int i = 0; i < 1000; i++ ) {
   if ( data[i] > largest ) {
     #pragma omp critical
     {
       if ( data[i] > largest ) largest = data[i];
     }
   }
}
```

Example: Find the Max from an Array

```
/* Third Edition */

int largest = 0;
int lp

#pragma omp parallel private(lp)
{
  lp = 0;
  #pragma omp for
  for ( int i = 0; i < 1000; i++) {
    if ( data[i] > lp )
      lp = data[i];
  }
  if ( lp > largest ) {
    #pragma critical
    {
      if ( lp > largest )
        largest = lp;
    }
  }
}
```

Example: Single and Master Directive

```c
#include <stdio.h>
extern float average(float,float,float);
void a15( float* x, float* xold, int n, float tol )
{
    int c, i, toobig;
    float error, y;
    c = 0;
    #pragma omp parallel
    {
        do{
            #pragma omp for private(i)
            for( i = 1; i < n-1; ++i ){
                xold[i] = x[i];
            }
            #pragma omp single
            {
                toobig = 0;
            }
            #pragma omp for private(i,y,error) reduction(+:toobig)
            for( i = 1; i < n-1; ++i ){
                y = x[i];
                x[i] = average( xold[i-1], x[i], xold[i+1] );
                error = y - x[i];
                if( error > tol || error < -tol ) ++toobig;
            }
            #pragma omp master
            {
                ++c;
                printf( "iteration %d, toobig=%d\n", c, toobig );
            }
        }while( toobig > 0 );
    }
}
```

OpenMP Library Functions

`void omp_set_num_threads (int num_threads);`

Sets the default number of threads that will be created on encountering the next parallel directive, provided the num_threads clause is not used in the parallel directive

`int omp_get_num_threads ();` Returns the number of threads in the team

`int omp_get_max_threads ();`

Returns the maximum number of threads that could possibly be created by a parallel directive encountered without the `num_threads` clause

`int omp_get_thread_num ();`

Returns a unique thread i.d. for each thread in a team

`int omp_get_num_procs ();`

Returns the number of processors that are available to execute the threaded program at that point

Example of `omp_set_num_threads()`

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

int main()
{
    omp_set_num_threads(4);
    printf_s("%d\n", omp_get_num_threads( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_get_num_threads( ));
        }

    printf_s("%d\n", omp_get_num_threads( ));

    #pragma omp parallel num_threads(3)
        #pragma omp master
        {
            printf_s("%d\n", omp_get_num_threads( ));
        }

    printf_s("%d\n", omp_get_num_threads( ));
}
```

```
Output:
1
4
1
3
1
```

OpenMP Library Functions

```
void omp_set_dynamic (int dynamic_threads);
```

Indicates that the number of threads available in subsequent parallel region can be adjusted by the run time.

If the value `dynamic_threads` evaluates to 0, such dynamic adjustment is disabled, otherwise it is enabled

```
int omp_get_dynamic ();
```

```
void omp_set_nested (int nested);
```

Enables nested parallelism (e.g., in recursive functions) if the value of its argument, nested, is non-zero, and disables it otherwise

```
int omp_get_nested ();
```

Environment Variables

OMP_NUM_THREADS

specifies the default number of threads for each parallel region.

OMP_DYNAMIC

when set to TRUE, allows the number of threads to be controlled at runtime using the omp_set_num_threads function or the num_threads clause.

OMP_NESTED

when set to TRUE, enables nested parallelism, unless it is disabled by calling the omp_set_nested function with a zero argument.

OMP_SCHEDULE

Controls the assignment of iteration spaces associated with for directives that use the *runtime* scheduling class

Precedences

Clause > Function > Environment Variable > Default

For example, the number of threads in a parallel region is determined by the following factors, in order of precedence:

- Evaluation of the IF clause
- Setting of the NUM_THREADS clause
- Use of the omp_set_num_threads() library function
- Setting of the OMP_NUM_THREADS environment variable
- Implementation default - usually the number of CPUs on a node

OpenMP 3.0: Task Constructs

```
#pragma omp task [clause ...]  newline
                   if (scalar expression)
                   final (scalar expression)
                   untied
                   default (shared | none)
                   mergeable
                   private (list)
                   firstprivate (list)
                   shared (list)

     structured_block
```

When a thread encounters a task construct, a task is generated from the code for the associated structured block.

The task will be executed by an encountering thread.

A task region binds to the current parallel team (the innermost enclosing parallel region)

OpenMP Task Constructs

```
// tasks
...
#pragma omp single nowait
{
   #pragma omp task
   foo();
   #pragma omp task
   bar();
}
#pragma omp taskwait
...
```

A single thread creates the tasks

`taskwait` is an (explicit) scheduling point

- When a thread hits the scheduling point, it will process tasks.
- Ensures that current execution flow will get paused until all queued tasks executed.

`nowait`

- Ensures the other threads will directly go to the scheduling point without waiting for the single thread that created the tasks

OpenMP Task Constructs

Equivalent code (with implicit Scheduling Point):

```
// tasks
...
#pragma omp single
{
   #pragma omp task
   foo();
   #pragma omp task
   bar();
}
...
```

Barrier synchronization is an implicit scheduling point

When a thread hits the scheduling point, it will process tasks.

OpenMP Task Constructs

If there are 3 threads, one possible scenario of what might happen is

```
                +--+-->[ task queue ]--+
                |  |                   |
                |  |        +-----------+
                |  |        |
Thread 0: --< single >-|   v  |-----
Thread 1: --------->|< foo() >|-----
Thread 2: --------->|< bar() >|-----
```

`|  ...  |` is the action at the scheduling point by each thread

In this case, thread 1 and thread 2 hit the scheduling point first.

OpenMP Task Constructs

Achieves essentially the same results as

```
#pragma omp sections
{
   #pragma omp section
   foo();
   #pragma omp section
   bar();
}
```

```
                 [    sections     ]
Thread 0: -------< section 1 >---->*------
Thread 1: -------< section 2      >*------
Thread 2: ------------------------>*------
...                                *
Thread N-1: ---------------------->*------
```

There is an implicit barrier at the end of the sections construct (shown here as *)

OpenMP Task Constructs

If there are 3 threads, another possible scenario is

```
               +--+-->[ task queue ]--+
               |  |                   |
               |  |      +------------+
               |  |      |
Thread 0: --< single >-| v             |---
Thread 1: --------->|< foo() >< bar() >|---
Thread 2: --------------------->|      |---
```

`|  ...  |` is the action at the scheduling point by each thread

In this case, thread 1 is able to finish processing the foo() task and request another one even before the other threads are able to request tasks.

OpenMP Task Constructs

Other possible scenarios:

```
               +--+-->[ task queue ]--+
               |  |                   |
               |  |      +------------+
               |  |      |
Thread 0: --< single >-| v < bar() >|---
Thread 1: --------->|< foo() >      |---
Thread 2: ----------------->|       |---

Thread 0: --< single: foo(); bar() >*---
Thread 1: ------------------------->*---
Thread 2: ------------>*---
```

In the second case, OpenMP runtime bypasses the task queue completely and executes the tasks serially.

OpenMP Tasks Constructs

Additional Resources:

Differences between Section and Task OpenMP

http://stackoverflow.com/questions/13788638/difference-between-section-and-task-openmp

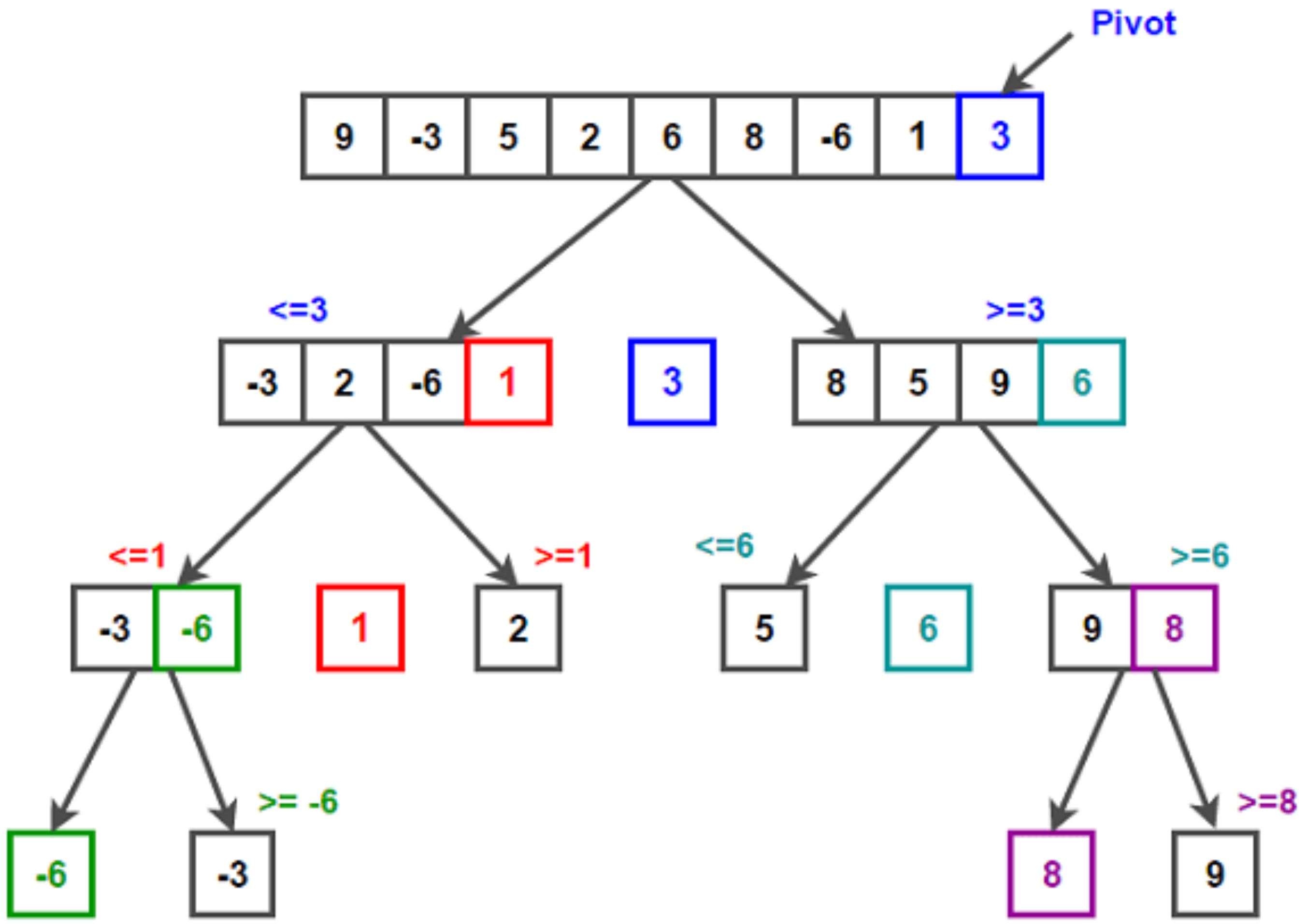
Useful Materials on eClass:

Taking Advantage of OpenMP 3.0 Tasking

Tasks vs. Sections

See the quickSort source code on eClass

- Sections vs. Tasks: which is faster?


Pivot
9 -3 5 2 6 8 -6 1 3
<=3
-3 2 -6 1
3
>=3
8 5 9 6
<=1
-3 -6
1
>=1
2
<=6
5
6
>=6
9 8
-6
>= -6
-3
8
>=8
9