CS3006 Parallel and Distributed Computing

FALL 2022

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Chapter 7. Programming Shared Address Space Platforms

Thread Basics

- A ***thread*** is a single stream of control in the flow of a program. A program like:

```
1    for (row = 0; row < n; row++)
2        for (column = 0; column < n; column++)
3            c[row][column] =
4                dot_product(get_row(a, row),
5                            get_col(b, col));
```

- The for loop in this code fragment has n^2 iterations, each of which can be executed independently. Such an independent sequence of instructions is referred to as a **thread**

- There are n^2 threads, one for each iteration of the for-loop.
- Since each of these threads can be executed independently of theothers, they can be scheduled **concurrently** on multiple processors.

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
            c[row][column] = create_thread(dot_product(get_row(a, row), get_col(b, col)));
```

- In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.
  - To execute the above code fragment on multiple processors, each processor must have access to matrices *a*, *b*, and *c*.

Thread Basics: Creation and Termination

```c
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

- In main() we declare a variable called **thread_id**, which is of type pthread_t, which is an integer used to identify the thread in the system.

- After declaring thread_id, we call **pthread_create()** function to create a thread. pthread_create() takes 4 arguments
  - The first argument is a pointer to thread_id which is set by this function.
  - The second argument specifies **attributes**. If the value is NULL, then default attributes shall be used.
  - The third argument is name of function to be executed for the thread to be created.
  - The fourth argument is used to **pass arguments** to the function, myThreadFun

- A call to `pthread_join` waits for the termination of the thread whose id is given by thread_id.
- On a successful call to pthread_join, the value passed to pthread_exit is returned in the location pointed to by the second argument.
- On successful completion, pthread_join returns 0, else it returns an error-code.

Synchronization Primitives in Pthreads

- When multiple threads attempt to manipulate the same data item, the results can often be incoherent if proper care is not taken to synchronize them.
- Consider:

```
/* each thread tries to update variable best_cost as follows */
if (my_cost < best_cost)
  best_cost = my_cost;
```

- Assume that there are two threads, the initial value of `best_cost` is 100, and the values of `my_cost` are **50** and **75** at threads t1 and t2.
- Depending on the schedule of the threads, the value of `best_cost` could be 50 or 75!
- The value 75 does not correspond to any serialization of the threads.

Mutual Exclusion

- The code in the previous example corresponds to a **critical segment**; i.e., a segment that must be executed by only one thread at any time.
- Critical segments in Pthreads are implemented using **Mutex locks** (mutual exclusion locks)
- Mutex-locks have two states: **locked** and **unlocked**. At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation.
- A thread entering a critical segment first tries to get a lock. It goes ahead when the lock is granted.
- If the mutex-lock is already locked, the process trying to acquire the lock is **blocked**.
  - This is because a locked mutex-lock implies that there is another thread currently in the critical section and that no other thread must be allowed in.

Mutual Exclusion

- The Pthreads API provides the following functions for handling mutex-locks:

```
int pthread_mutex_lock(pthread_mutex_t *mutex_lock);
```

  - On leaving a critical section, a thread must unlock the mutex-lock associated with the section. If it does not do so, **no other** thread will be able to enter this section. Typically **deadlock**.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex_lock);
```

  - On calling this function, in the case of a normal mutex-lock, the lock is relinquished and one of the blocked threads is scheduled to enter the critical section.

Mutual Exclusion

- If a programmer attempts a **pthread_mutex_unlock** on a previously unlocked mutex or one that is locked by another thread, the effect is undefined.

OVERHEADS OF LOCKING

- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can, therefore, lead to significant performance degradation.
- It is important to minimize the size of critical sections.

```
1   void *find_entries(void *start_pointer) {
2
3       /* This is the thread function */
4
5       struct database_record *next_record;
6       int count;
7       current_pointer = start_pointer;
8       do {
9           next_record = find_next_entry(current_pointer);
10          count = output_record(next_record);
11      } while (count < requested_number_of_records);
12  }
13
14  int output_record(struct database_record *record_ptr) {
15      int count;
16      pthread_mutex_lock(&output_count_lock);
17      output_count ++;
18      count = output_count;
19      pthread_mutex_unlock(&output_count_lock);
20
21      if (count <= requested_number_of_records)
22          print_record(record_ptr);
23      return (count);
24  }
```

Example: Finding k matches in a list

Alleviating Locking Overheads

- It is often possible to reduce the idling overhead associated with locks using an alternate function, **pthread_mutex_trylock**.

- This function attempts a lock on **mutex_lock**. If the lock is successful, the function returns a *zero*. If it is already locked by another thread, instead of blocking the thread execution it returns a value EBUSY.

- This allows the thread to do other work and to poll the mutex for a lock.

```
1    int output_record(struct database_record *record_ptr) {
2        int count;
3        int lock_status;
4         lock_status = pthread_mutex_trylock(&output_count_lock);
5        if (lock_status == EBUSY) {
6             insert_into_local_list(record_ptr);
7            return(0);
8        }
9        else {
10            count = output_count;
11            output_count += number_on_local_list + 1;
12            pthread_mutex_unlock(&output_count_lock);
13            print_records(record_ptr, local_list,
14                  requested_number_of_records - count);
15            return(count + number_on_local_list + 1);
16       }
17   }
```

- While the function **pthread_mutex_trylock** alleviates this overhead, it introduces the overhead of **polling** for availability of locks.
  - threads would have to periodically *poll* for availability of lock.
- A natural solution to this problem is an **interrupt driven** mechanism as opposed to a polled mechanism.
- The availability of space is signaled by the consumer thread that consumes the task.
- The functionality to accomplish this is provided by a ***condition variable***.
- A condition variable is a data object used for synchronizing threads. This variable allows a thread to block itself until specified data reaches a predefined state.

OpenMP: a Standard for Directive Based Parallel Programming

- OpenMP is a directive-based API that can be used with FORTRAN, C, and C++ for programming shared address space machines.

- OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

- OpenMP directives in C and C++ are based on the `#pragma` compiler directives.

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

- OpenMP programs execute serially until they encounter the parallel directive.

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

- This directive is responsible for creating a group of threads.
  - The exact number of threads can be specified in the directive, set using an environment variable, or at runtime using **OpenMP** functions.
- 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.

- Each thread created by this directive executes the structured block specified by the parallel directive.

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

- The **clause list** is used to specify conditional parallelization, number of threads, and data handling.

- **Conditional Parallelization:** The clause `if (scalar expression)` determines whether the parallel construct results in creation of threads.
  - Only one if clause can be used with a parallel directive.

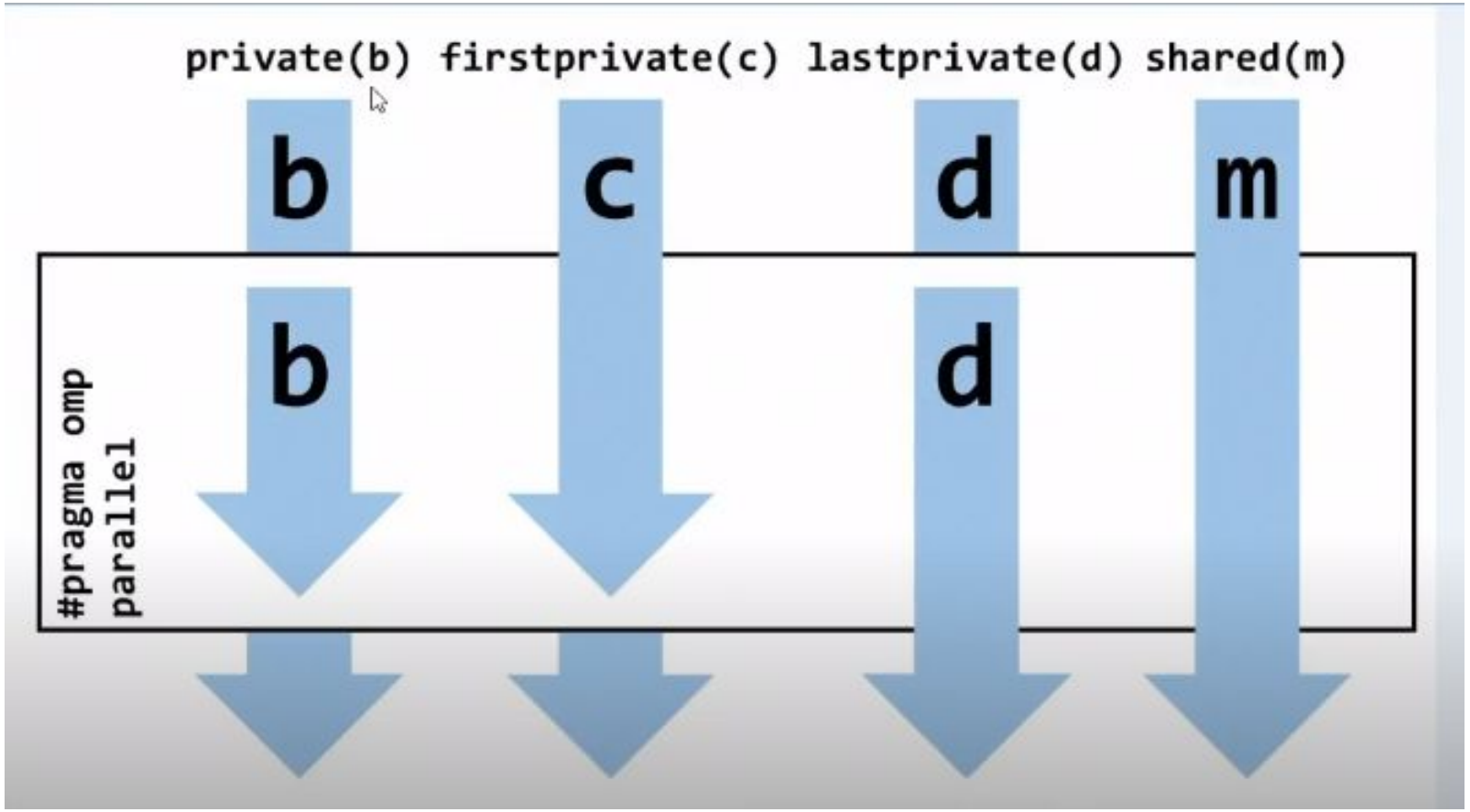
- **Degree of Concurrency:** The clause `num_threads (integer expression)` specifies the number of threads that are created by the parallel directive.

Data Handling

- The clause **`private (variable list)`** indicates that the set of variables specified is local to each thread – i.e., each thread has its own copy of each variable in the list.

- The clause **`firstprivate (variable list)`** is similar to the private clause, except the values of variables on entering the threads are initialized to corresponding values before the parallel directive.

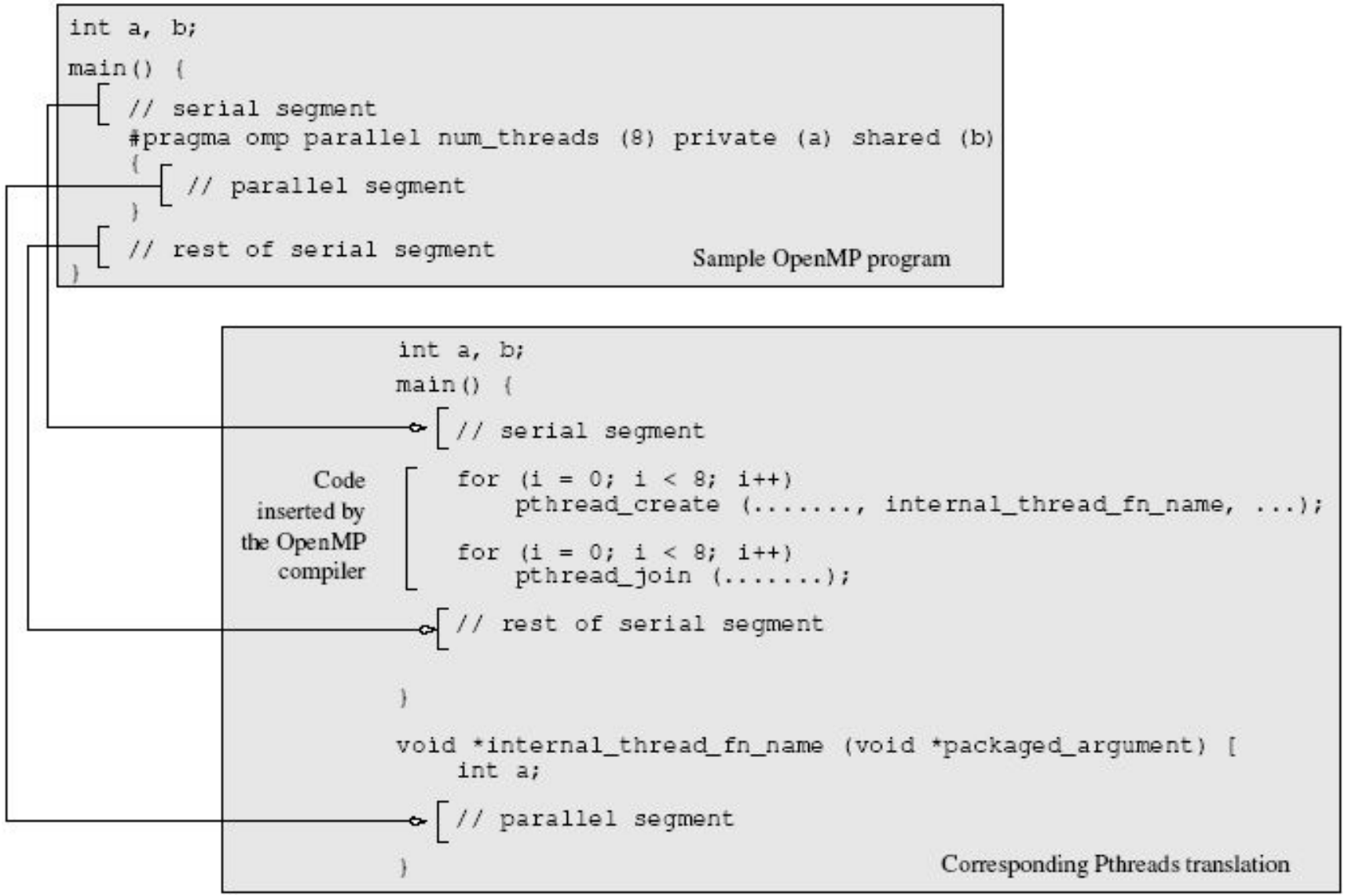
- The clause **`shared (variable list)`** indicates that all variables in the list are shared across all the threads, i.e., there is only one copy. Special care must be taken while handling these variables by threads to ensure serializability.


private(b) firstprivate(c) lastprivate(d) shared(m)
b
c
d
m
b
d
#pragma omp
parallel

Using the parallel directive

```
1    #pragma omp parallel if (is_parallel == 1) num_threads(8) \
2                         private (a) shared (b) firstprivate(c)
3    {
4        /* structured block */
5    }
```

- Here, if the value of the variable is_parallel equals one, eight threads are created.
- Each of these threads gets private copies of variables a and c, and shares a single value of variable b.
- Furthermore, the value of each copy of c is initialized to the value of c before the parallel directive.



A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

Clause	Description
`private`	Declares variables to be `private` to each thread in a team. Private copies of the variable are initialized from the original object when entering the region.
`firstprivate`	Provides a superset of the functionality provided by the `private` clause. Each private data object is initialized with the value of the original object.
`lastprivate`	Provides a superset of the functionality provided by the `private` clause. The original object is updated with the value of the private copy from the last sequential iteration of the associated loop, or the lexically last section construct, when exiting the region.
`shared`	Shares variables among all the threads in a team.
`default`	Enables you to affect the data-scope attributes of variables.
`reduction`	Performs a reduction on scalar variables.
`ordered`	The structured block following an `ordered` directive is executed in the order in which iterations would be executed in a sequential loop.

Clause	Description
if (*expression*)	If the `if(expression)` clause is present, the enclosed code block is executed in parallel only if the *expression* evaluates to `TRUE`. Otherwise the code block is serialized. The expression must be scalar logical.
`schedule`	Specifies how iterations of the `for` loop are divided among the threads of the team.
collapse(*n*)	Specifies how many loops are associated with the OpenMP loop construct for collapsing.
`copyin`	Provides a mechanism to copy the data values of the master thread to the variables used by the `threadprivate` copies at the beginning of the parallel region.
`copyprivate`	Provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region.
`nowait`	Indicates that an implementation may omit the barrier at the end of the worksharing region.
`untied`	Indicates that a resumed task does not have to be executed by same thread executing it before it was suspended.
`mergeable`	Indicates that the task defined by this task pragma need not create a private data environment for the task, but if the task execution is deferred, then the private data environment is created.
`final`(*expr*)	The *expr* is evaluated, and if the value is true, then this task and all its descendant tasks are non-deferred (not executed in parallel).

- The default state of a variable is specified by the clause `default(shared)` or `default (none)`.
  - The clause `default (shared)` implies that, by default, a variable is shared by all the threads.
  - The clause `default (none)` implies that the state of each variable used in a thread must be explicitly specified. This is generally recommended, to guard against errors arising from unintentional concurrent access to shared data.

reduction clause

- Just as `firstprivate` specifies how multiple local copies of a variable are initialized inside a thread, the `reduction` clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.

- The OpenMP `reduction` clause lets you specify one or more thread-*private variables* that are subject to a reduction operation at the end of the parallel region.
  - OpenMP predefines a set of reduction operators. Each reduction variable must be a **scalar** (for example, int, long, and float).
  - OpenMP also defines several restrictions on how reduction variables are used in a parallel region.

- The operator can be one of +, *, -, &, |, ^, &&, and ||

```
1.  #pragma omp parallel reduction(+: sum) num_threads(8)
2.    {
3.          /* compute local sums here */
4.    }
5.  /* sum here contains sum of all local instances of sums
6.  */
```

- In this example, each of the eight threads gets a copy of the variable **sum**.
- When the threads exit, the sum of all of these local copies is stored in the single copy of the variable (at the **master thread**) .

```
1    /*  ************************************************************
2      An OpenMP version of a threaded program to compute PI.
3       ************************************************************ */
4
5       #pragma omp parallel default(private) shared (npoints) \
6                                reduction(+: sum) num_threads(8)
7       {
8          num_threads = omp_get_num_threads();
9          sample_points_per_thread = npoints / num_threads;
10         sum = 0;
11         for (i = 0; i < sample_points_per_thread; i++) {
12            rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
13            rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
14           if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
15               (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
16              sum ++;
17         }
18      }
```

Specifying Concurrent Tasks in OpenMP

- The parallel directive can be used in conjunction with other directives to specify concurrency across iterations and tasks.

- OpenMP provides two directives – **`for`** and **`sections`** – to specify concurrent iterations and tasks.

The `for` Directive

- The for directive is used to split parallel iteration spaces across threads. The general form of a for directive is as follows:

```
1.  #pragma omp for [clause list]
2.        /* for loop */
```

- The clauses that can be used in this context are: `private`, `firstprivate`, `lastprivate`, `reduction`, `schedule`, `nowait`, and `ordered`.

The `for` Directive

- When using a for loop for farming work to threads, it is sometimes desired that the last iteration (as defined by serial execution) of the for loop update the value of a variable.

- This is accomplished using the `lastprivate` directive.

- The `lastprivate` clause deals with how multiple local copies of a variable are written back into a single copy at the end of the parallel for loop.

Assigning Iterations to Threads

- The `schedule` clause of the for directive deals with the assignment of iterations to threads.

- The general form of the schedule directive is `schedule(scheduling_class[, parameter])`.

- OpenMP supports four scheduling classes: `static`, `dynamic`, `guided`, and `runtime`.

Assigning Iterations to Threads

- The general form of the **static scheduling** class is `schedule(static[,chunk-size])`.
  - E.g. The following for loop will be divided in 4 chunks, each with 5 iterations:

```
#pragma omp parallel for schedule(static, 5) num_threads(4)
for (i = 0; i < 20; i++)
```

- When no chunk-size is specified, the total iterations are split into as many chunks as there are threads and one chunk is assigned to each thread.

The following modification of the matrix-multiplication program causes the outermost iteration to be split statically across threads as illustrated in Figure 7.5(a).

```
1    #pragma omp parallel default(private) shared (a, b, c, dim) \
2                         num_threads(4)
3        #pragma omp for schedule(static)
4        for (i = 0; i < dim; i++) {
5            for (j = 0; j < dim; j++) {
6                c(i,j) = 0;
7                for (k = 0; k < dim; k++) {
8                    c(i,j) += a(i, k) * b(k, j);
9                }
10           }
11       }
```

Dynamic

- Often, because of a number of reasons, like heterogeneous computing resources to non-uniform processor loads, equally partitioned workloads take widely varying execution times.
- For this reason, OpenMP has a dynamic scheduling class.
- The general form of this class is `schedule(dynamic[, chunk-size])`
- The iteration space is partitioned into chunks given by chunk-size. However, these are assigned to threads as they become idle.
- This takes care of the temporal imbalances resulting from static scheduling. If no chunk-size is specified, it defaults to a single iteration per chunk.

```
// Enable OpenMP in Properties -> C / C++ -> Language
#include "stdafx.h"
#include <iostream>
#include <omp.h>
#include <stdio.h>

#define N 16

using namespace std;

void printVar(int a[], int b, int c, int d, int m) {
cout << "b = " << b << ", c = " << c << ", d = " << d << " m = " << m << endl;
cout << "a = ";
for (int i = 0; i < N; i++)
cout << a[i] << ", ";
cout << endl;
}
```

```
int main(){
int i;
int a[N];
int b = 0, c = 0, d = 0, m = 0;
cout << "Before par. for" << endl;
printVar(a, b, c, d, m); //Garbage Values

#pragma omp parallel for default(none) private(i,b) firstprivate(c) lastprivate(d) shared(m,a)
schedule(static, 3) num_threads(4)
for (i = 0; i < N; i++) {
// b and d must be initialized
b = 100;
d = 10;
printf("Thread %d, iteration %d: b = %d, c = %d, d = %d, m = %d\n", omp_get_thread_num(), i, b,
c, d, m);
a[i] = omp_get_thread_num();
b = omp_get_thread_num();
c = omp_get_thread_num();
d = omp_get_thread_num();
m = omp_get_thread_num();
}
cout << "After par. for" << endl;
printVar(a, b, c, d, m);
getchar();
return 0;
}
```

Guided

- Consider the partitioning of an iteration space of 100 iterations with a chunk size of 5. This corresponds to 20 chunks. If there are 16 threads, in the best case, 12 threads get one chunk each and the remaining four threads get two chunks. this assignment results in considerable idling.

- The solution to this problem (also referred to as an ***edge effect***) is to reduce the chunk size as we proceed through the computation. This is the principle of the guided scheduling class.

- The general form of this class is `schedule(guided[, chunk-size])`.

- In this class, the chunk size is reduced exponentially as each chunk is dispatched to a thread. The chunk-size refers to the smallest chunk that should be dispatched.

Runtime

- Often it is desirable to delay scheduling decisions until runtime.
- For example, if one would like to see the impact of various scheduling strategies to select the best one, the scheduling can be set to runtime.
- In this case the environment variable `OMP_SCHEDULE` determines the scheduling class and the chunk size.

Class Tasks

- Calculate average of 100 elements' integer array using reduction clause. Use 8 threads only, also display the local results of every thread with its ID.

- Write some program to differentiate between 'STATIC' and 'DYNAMIC' scheduling with parallel for.

- Write a Parallel C++ program to calculate WORDCOUNT of some text file. Your program must have reduction clause, and also display the local results of each thread.