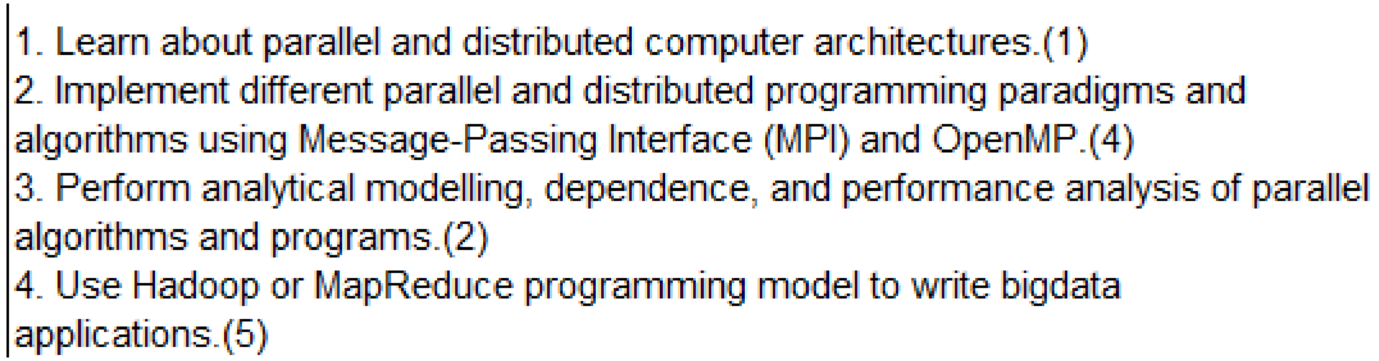
CS 3006 Parallel and Distributed Computer

Fall 2022



1. Learn about parallel and distributed computer architectures.(1)
2. Implement different parallel and distributed programming paradigms and algorithms using Message-Passing Interface (MPI) and OpenMP.(4)
3. Perform analytical modelling, dependence, and performance analysis of parallel algorithms and programs.(2)
4. Use Hadoop or MapReduce programming model to write bigdata applications.(5)

Week # 7 – Lecture # 16, 17, 18

6th, 8th, 9th Rabi ul Awwal, 1444

3rd , 5th , 6th October 2022

Dr. Nadeem Kafi Khan

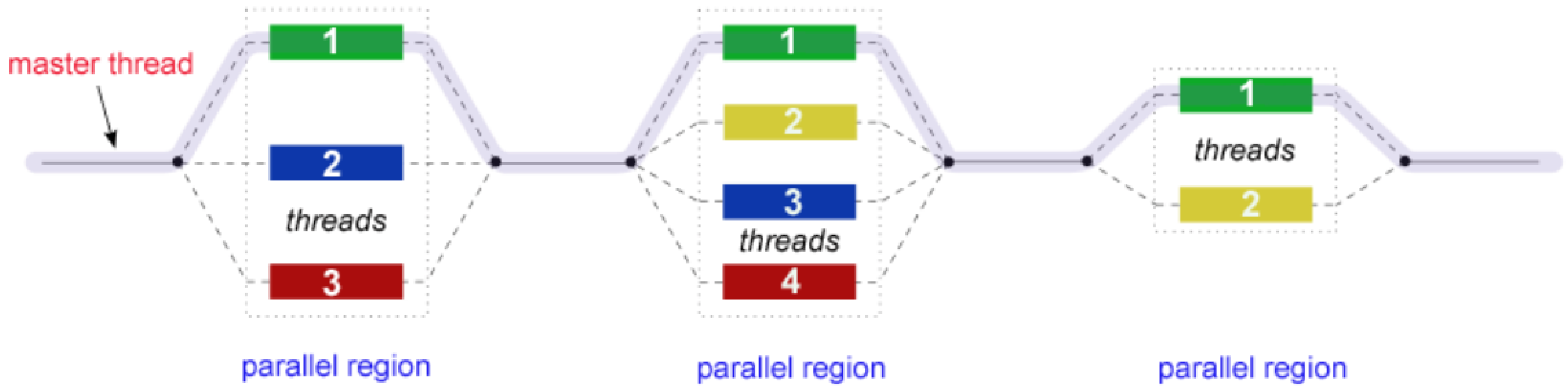
Lecture # 16 – Topics (Lab # 7)

- OpenMP Solution stack and Parallel execution model
- Hello, World! OpenMP Program
- Type of Parallelism with OpenMP
- Paralleling Serial Pi Program with OpenMP
    - First attempt
    - False Sharing Problem
    - 2nd attempt without False Sharing Problem
    - 3rd attempt without storing results in a Global Array
- #pragma atomic
- What is the difference in using #pragma critical vs #pragma atomic

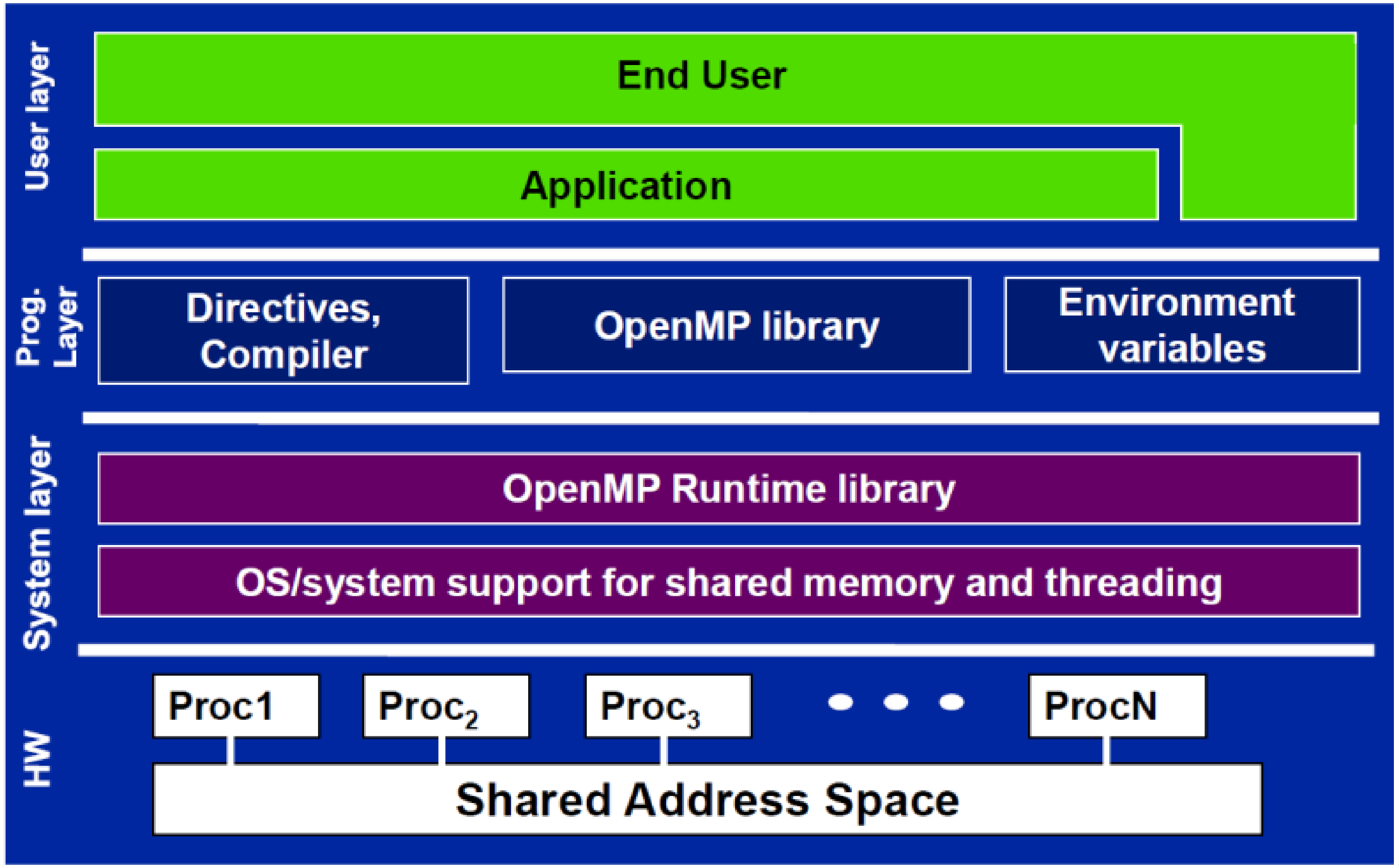
Key Concepts in OpenMP

- Parallel regions where parallel execution occurs via multiple concurrently executing threads
  - Each thread has its own program counter and executes one instruction at a time, similar to sequential program execution
- Shared and private data: shared variables are the means of communicating data between threads
- Synchronization: fundamental means of coordinating execution of concurrent threads
- Mechanism for **automated work distribution** across threads

Fork-Join Model of Parallel Execution



OpenMP Solution Stack



The OpenMP API

- Compiler directives
  - `#pragma omp parallel`
  - Comments, ignored unless instructed not to
- Runtime library routines
  - `int omp_get_num_threads(void);`
- Environment variables
  - `export OMP_NUM_THREADS=8`

Hello World with OpenMP!

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

using namespace std;

int main() {
  cout << "This is serial code\n";
#pragma omp parallel
  {
    int num_threads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    if (tid == 0) {
      cout << num_threads << "\n";
    }
    cout << "Hello World: " << tid << "\n";
  }

  cout << "This is serial code\n";
#pragma omp parallel num_threads(2)
  {
    int tid = omp_get_thread_num();
    cout << "Hello World: " << tid << "\n";
  }

  cout << "This is serial code\n";

  omp_set_num_threads(3);
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    cout << "Hello World: " << tid << "\n";
  }
}
```

- Each thread has a unique integer “id”; master thread has “id” 0, and other threads have “id” 1, 2, ...
- OpenMP runtime function omp_get_thread_num() returns a thread’s unique “id”
- The function omp_get_num_threads() returns the total number of executing threads
- The function omp_set_num_threads(x) asks for “x” threads to execute in the next parallel region (must be set outside region)

Types of Parallelism with OpenMP

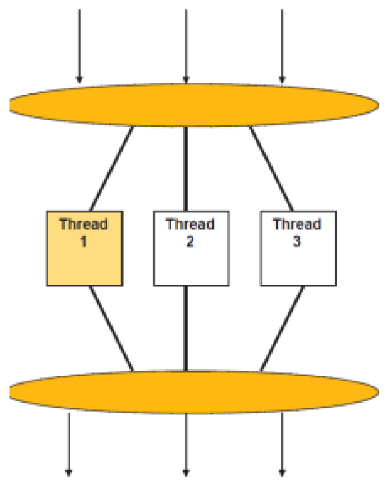
Coarse-grained

- Task parallelism
  - Split the work among threads that execute in parallel
  - Implicit join at the end of the segment, or explicit synchronization points

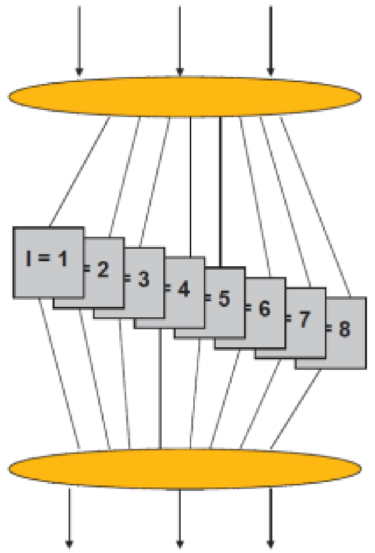
Fine-grained

- Loop parallelism
  - Execute independent iterations of for-loops in parallel
  - Several choices in splitting the work

Task parallelism



Loop parallelism



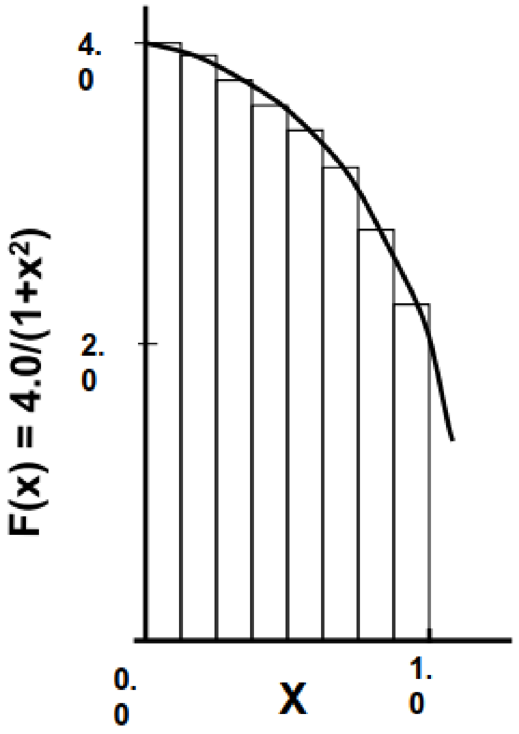
The Essence of OpenMP

- **Create threads that execute in a shared address space**
  - The only way to create threads is with the `parallel` construct
  - Once created, all threads execute the code inside the construct
- **Split up the work between threads by one of two means**
  - SPMD (Single Program Multiple Data) – all threads execute the same code and you use the thread ID to assign work to a thread
  - Workshare constructs split up loops and tasks between threads
- **Manage data environment to avoid data access conflicts**
  - Synchronization so correct results are produced regardless of how threads are scheduled
  - Carefully manage which data can be private (local to each thread) and shared

Recurring Example of Numerical Integration

Serial Pi Program

```
double seq_pi() {
  int i;
  double x, pi, sum = 0.0;
  double step = 1.0 / (double)NUM_STEPS;
  for (i = 0; i < NUM_STEPS; i++) {
    x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
  }
  pi = step * sum;
  return pi;
}
```



- Mathematically

$$\int_0^1 \frac{4}{(1+x^2)} dx = \pi$$

- We can approximate the integral as the sum of the rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

where each rectangle has width Δx and height $F(x_i)$ at the middle of interval *i*

Computing Pi with OpenMP

```
double omp_pi_with_fs() {
  omp_set_num_threads(NUM_THRS);
  double sum[NUM_THRS] = {0.0};
  double pi = 0.0;
  double step = 1.0 / (double)NUM_STEPS;
  uint16_t num_thrs;
```

```
#pragma omp parallel
  {
    // Parallel region with worker threads
    uint16_t tid = omp_get_thread_num();
    uint16_t nthrds = omp_get_num_threads();
    if (tid == 0) {
      num_thrs = nthrds;
    }
    double x;
    for (int i = tid; i < NUM_STEPS; i += nthrds) {
      x = (i + 0.5) * step;
      sum[tid] += 4.0 / (1.0 + x * x);
    }
  } // end #pragma omp parallel
  for (int i = 0; i < num_thrs; i++) {
    pi += (sum[i] * step);
  }
  return pi;
}
```

Avoid False Sharing

- Array `sum[]` is a shared array, with each thread accessing exactly on element
- Cache line holding multiple elements of sum will be locally cached by each processor in its private L1 cache
- When a thread writes into into an index in `sum`, the entire cache line becomes “dirty” and causes invalidation of that line in all other processor’s caches
- Cache thrashing due to this “false sharing” causes performance degradation

Parallel computation with padding

```
double omp_pi_without_fs1() {
  omp_set_num_threads(NUM_THRS);
  double sum[NUM_THRS][8];
  double pi = 0.0;
  double step = 1.0 / (double)NUM_STEPS;
  uint16_t num_thrs;
#pragma omp parallel
  {
    uint16_t tid = omp_get_thread_num();
    uint16_t nthrds = omp_get_num_threads();
    if (tid == 0) {
      num_thrs = nthrds;
    }
    double x;
    for (int i = tid; i < NUM_STEPS; i += nthrds) {
      x = (i + 0.5) * step;
      sum[tid][0] += 4.0 / (1.0 + x * x);
    }
  } // end #pragma omp parallel

  for (int i = 0; i < num_thrs; i++) {
    pi += (sum[i][0] * step);
  }
  return pi;
}
```

Parallel computation with thread-local sum

```
double omp_pi_without_fs2() {
  omp_set_num_threads(NUM_THRS);
  double pi = 0.0;
  double step = 1.0 / (double)NUM_STEPS;
  uint16_t num_thrs;
```

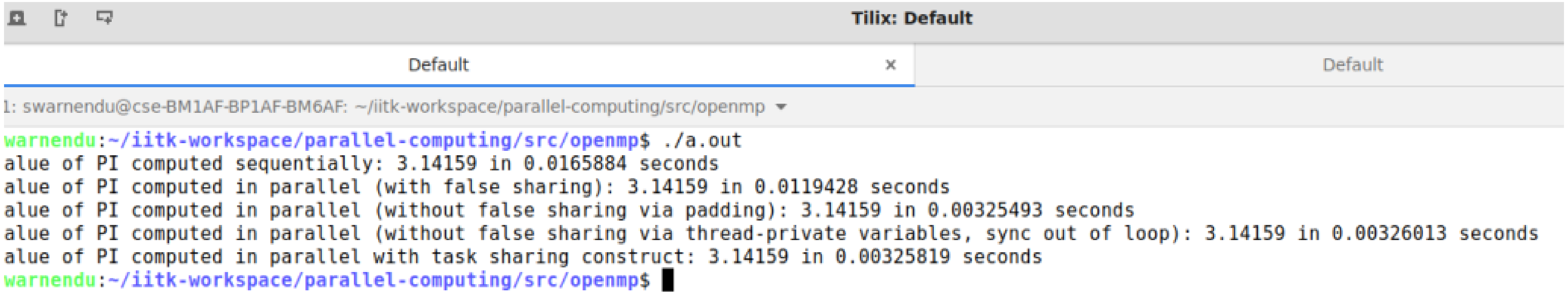
What the problem in this code?

```
#pragma omp parallel
  {
    uint16_t tid = omp_get_thread_num();
    uint16_t nthrds = omp_get_num_threads();
    if (tid == 0) {
      num_thrs = nthrds;
    }
    double x, sum;
    for (int i = tid; i < NUM_STEPS; i += nthrds) {
      x = (i + 0.5) * step;
      // Scalar variable sum is
      // thread-private, so no false sharing
      sum += 4.0 / (1.0 + x * x);
    }
  #pragma omp critical // Mutual exclusion
    pi += (sum * step);
  } // end #pragma omp parallel

  return pi;
}
```

Evaluate the Pi Program Variants

- Sequential computation of pi
- Parallel computation with false sharing
- Parallel computation with padding
- Parallel computation with thread-local sum


Tilix: Default
Default
Default
1: swarnendu@cse-BM1AF-BP1AF-BM6AF: ~/iitk-workspace/parallel-computing/src/openmp

```
warnendu:~/iitk-workspace/parallel-computing/src/openmp$ ./a.out
alue of PI computed sequentially: 3.14159 in 0.0165884 seconds
alue of PI computed in parallel (with false sharing): 3.14159 in 0.0119428 seconds
alue of PI computed in parallel (without false sharing via padding): 3.14159 in 0.00325493 seconds
alue of PI computed in parallel (without false sharing via thread-private variables, sync out of loop): 3.14159 in 0.00326013 seconds
alue of PI computed in parallel with task sharing construct: 3.14159 in 0.00325819 seconds
warnendu:~/iitk-workspace/parallel-computing/src/openmp$ 
```

`atomic` Construct

- Atomic is an efficient critical section for simple reduction operations
- Applies only to the update of a memory location
- Uses hardware atomic instructions for implementation; much lower overhead than using critical section

```
float res;
#pragma omp parallel
{
  float B;
  int id = omp_get_thread_num();
  int nthrds = omp_get_num_threads();
  for (int i = id; i < MAX; i += nthrds) {
    B = big_job(i);
#pragma omp atomic
    res += B;
  }
}
```

Often, a critical section consists simply of an update to a single memory location, for example, incrementing or adding to an integer. OpenMP provides another directive, `atomic`, for such atomic updates to memory locations. The `atomic` directive specifies that the memory location update in the following instruction should be performed as an atomic operation. The update instruction can be one of the following forms:

```
x binary_operation = expr
x++
++x
x--
--x
```

Here, `expr` is a scalar expression that does not include a reference to `x`, `x` itself is an lvalue of scalar type, and `binary_operation` is one of {+, *, -, /, &, ·||, ≪, ≫,}. It is important to note that the `atomic` directive only atomizes the load and store of the scalar variable. The evaluation of the expression is not atomic. Care must be taken to ensure that there are no race conditions hidden therein. This also explains why the `expr` term in the `atomic` directive cannot contain the updated variable itself. All `atomic` directives can be replaced by `critical` directives provided they have the same name. However, the availability of atomic hardware instructions may optimize the performance of the program, compared to translation to `critical` directives.

Atomic

- **If your critical section is of the form**

  `x += y` **OR** **`x = x <operation> y`**
- **Compiler can use hardware atomic operations**
- **Think "mini-critical"**

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

`critical` vs `atomic`

`critical`	**`atomic`**
• Locks code segments	• Locks data variables
• Serializes all unnamed critical sections	• Serializes operations on the same shared data
• Less efficient than `atomic`	• Makes use of hardware instructions to provide atomicity
• More general	• Less general

Lecture # 17 – Topics

- Coverage of all topics of Section 7.10 from the textbook.

7.10 OpenMP: a Standard for Directive Based Parallel Programming

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

```

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

Example 7.9 Using the parallel directive

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

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. ■

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.

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 usage of the `reduction` clause is `reduction (operator: variable list)`. This clause performs a reduction on the scalar variables specified in the list using the `operator`. The variables in the list are implicitly specified as being private to threads. The `operator` can be one of `+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`.

Example 7.10 Using the reduction clause

```
#pragma omp parallel reduction(+: sum) num_threads(8)
{
    /* compute local sums here */
}
/* sum here contains sum of all local instances of sums */
```

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). ■

`reduction` clause is `reduction (operator: variable list)`. This clause performs a reduction on the scalar variables specified in the list using the `operator`. The variables in the list are implicitly specified as being private to threads. The `operator` can be one of `+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`.

Reduction operators

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.

```
reduction(<operator>: <variable list>)
```

$\hookrightarrow$ +, *, -, &, |, ^, &&, ||

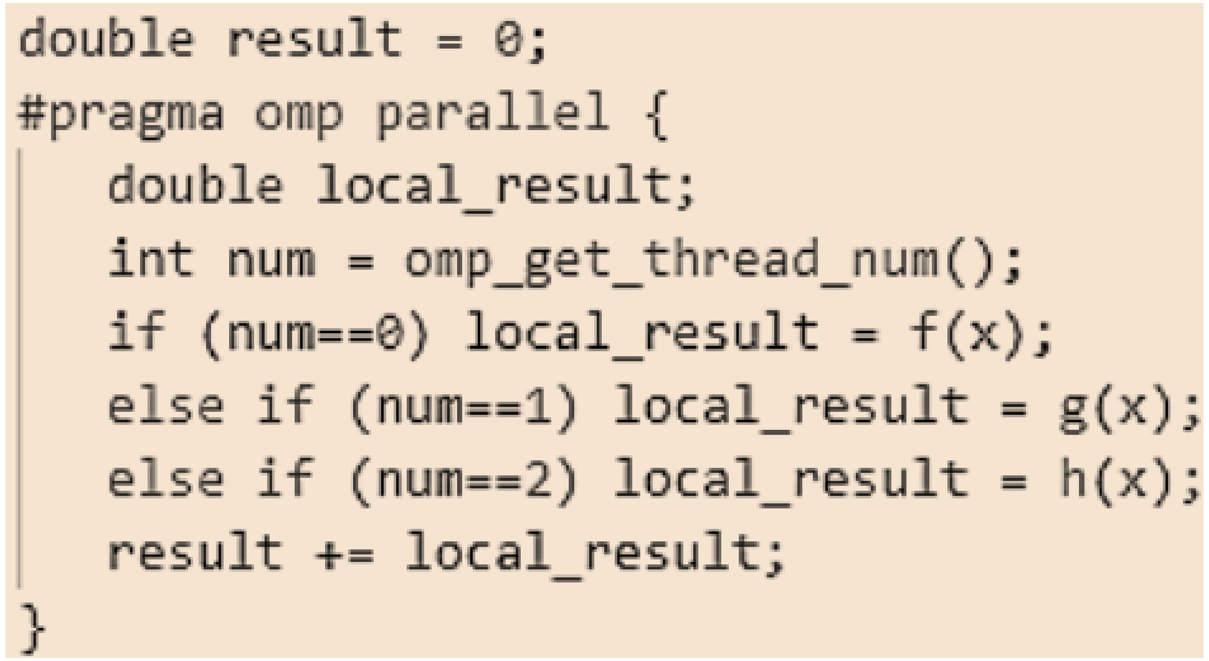
```
global_result = 0.0; // note that this variable is before the parallel clause.
#pragma omp parallel reduction(+: global_result)
global_result += Trap(a, b, n);
```

Parallel tasks often produce some quantity that needs to be summed together. One such condition exists in the code shown in Figure 1 where result variable is summed.

i. State the problem in the code (one sentence only), and

ii. Show new code by adding/correcting statements which produces correct results.

Use OMP constructs and clauses only.



```
double result = 0;
#pragma omp parallel {
    double local_result;
    int num = omp_get_thread_num();
    if (num==0) local_result = f(x);
    else if (num==1) local_result = g(x);
    else if (num==2) local_result = h(x);
    result += local_result;
}
```

i) Multiple threads will try to update variable "result" by adding their private variable "local_result" overwriting previous value.

ii) Any one of the following:
#pragma omp parallel reduction (+: result)
OR
#pragma omp critical
result +=local_result;

Example 7.11 Computing PI using OpenMP directives

```
/* ******************************************************
   An OpenMP version of a threaded program to compute PI.
   ****************************************************** */

   #pragma omp parallel default(private) shared (npoints) \
                            reduction(+: sum) num_threads(8)
   {
     num_threads = omp_get_num_threads();
     sample_points_per_thread = npoints / num_threads;
     sum = 0;
     for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
       if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
           sum ++;
     }
   }
```

Example 7.12 Using the for directive for computing π

```
    #pragma omp parallel default(private) shared (npoints) \
                         reduction(+: sum) num_threads(8)
    {
      sum=0;
      #pragma omp for
      for (i = 0; i < npoints; i++) {
          rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
          rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
             (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum ++;
      }
    }
```

Example 7.15 Using the nowait clause

Consider the following example in which variable name needs to be looked up in two lists – current_list and past_list. If the name exists in a list, it must be processed accordingly. The name might exist in both lists. In this case, there is no need to wait for all threads to complete execution of the first loop before proceeding to the second loop. Consequently, we can use the nowait clause to save idling and synchronization overheads as follows:

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i = 0; i < nmax; i++)
            if (isEqual(name, current_list[i])
                processCurrentName(name);
    #pragma omp for
        for (i = 0; i < mmax; i++)
            if (isEqual(name, past_list[i])
                processPastName(name);
}
```

The sections Directive

The for directive is suited to partitioning iteration spaces across threads. Consider now a scenario in which there are three tasks (taskA, taskB, and taskC) that need to be executed. Assume that these tasks are independent of each other and therefore can be assigned to different threads. OpenMP supports such non-iterative parallel task assignment using the sections directive. The general form of the sections directive is as follows:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

This sections directive assigns the structured block corresponding to each section to one thread (indeed more than one section can be assigned to a single thread). The clause list may include the following clauses – private, firstprivate, lastprivate, reduction, and no wait. The syntax and semantics of these clauses are identical to those in the case of the for directive. The lastprivate clause, in this case, specifies that the last section (lexically) of the sections directive updates the value of the variable. The nowait clause specifies that there is no implicit synchronization among all threads at the end of the sections directive.

Sections

- Sections will be executed in parallel
- Can combine parallel and section like we did with for
- If more threads than sections then idle threads will exist
- If less threads than sections then some sections will execute in serial

For executing the three concurrent tasks `taskA`, `taskB`, and `taskC`, the corresponding `sections` directive is as follows:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

Merging Directives

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        taskA();
    }
    #pragma omp section
    {
        taskB();
    }
    /* other sections here */
}
```

Sections

```
#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 */
}
```

Nesting `parallel` Directives

```
#pragma omp parallel for default(private) shared (a, b, c, dim) \
                    num_threads(2)
    for (i = 0; i < dim; i++) {
    #pragma omp parallel for default(private) shared (a, b, c, dim) \
                    num_threads(2)
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            #pragma omp parallel for default(private) \
                    shared (a, b, c, dim) num_threads(2)
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

Nesting `parallel` Directives

We start by making a few observations about how this segment is written. Instead of nesting three `for` directives inside a single `parallel` directive, we have used three `parallel for` directives. This is because OpenMP does not allow `for`, `sections`, and `single` directives that bind to the same `parallel` directive to be nested. Furthermore, the code as written only generates a logical team of threads on encountering a nested `parallel` directive. The newly generated logical team is still executed by the same thread corresponding to the outer `parallel` directive. To generate a new set of threads, nested parallelism must be enabled using the `OMP_NESTED` environment variable. If the `OMP_NESTED` environment variable is set to `FALSE`, then the inner `parallel` region is serialized and executed by a single thread. If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled. The default state of this environment variable is `FALSE`, i.e., nested parallelism is disabled. OpenMP environment

Synchronization Point: The `barrier` Directive

A barrier is one of the most frequently used synchronization primitives. OpenMP provides a `barrier` directive, whose syntax is as follows:

```
      #pragma omp barrier
```

- **When a thread executes a barrier it will wait until all other threads in the team also execute the barrier.**
- **Then threads continue working as usual.**

On encountering this directive, all threads in a team wait until others have caught up, and then release. When used with nested `parallel` directives, the `barrier` directive binds to the closest `parallel` directive. For executing barriers conditionally, it is important to note that a `barrier` directive must be enclosed in a compound statement that is conditionally executed. This is because pragmas are compiler directives and not a part of the language. Barriers can also be effected by ending and restarting `parallel` regions. However, there is usually a higher overhead associated with this. Consequently, it is not the method of choice for implementing barriers.

Single Thread Executions: The `single` and `master` Directives

A `single` directive specifies a structured block that is executed by a single (arbitrary) thread. The syntax of the `single` directive is as follows:

```
#pragma omp single [clause list]
             structured block
```

The clause list can take clauses `private`, `firstprivate`, and `nowait`. These clauses have the same semantics as before. On encountering the `single` block, the first thread enters the block. All the other threads proceed to the end of the block. If the `nowait` clause has been specified at the end of the block, then the other threads proceed; otherwise they wait at the end of the `single` block for the thread to finish executing the block. This directive is useful for computing global data as well as performing I/O.

Single Thread Executions: The `single` and `master` Directives

The `master` directive is a specialization of the `single` directive in which only the master thread executes the structured block. The syntax of the `master` directive is as follows:

```
#pragma omp master
          structured block
```

In contrast to the `single` directive, there is no implicit barrier associated with the `master` directive.

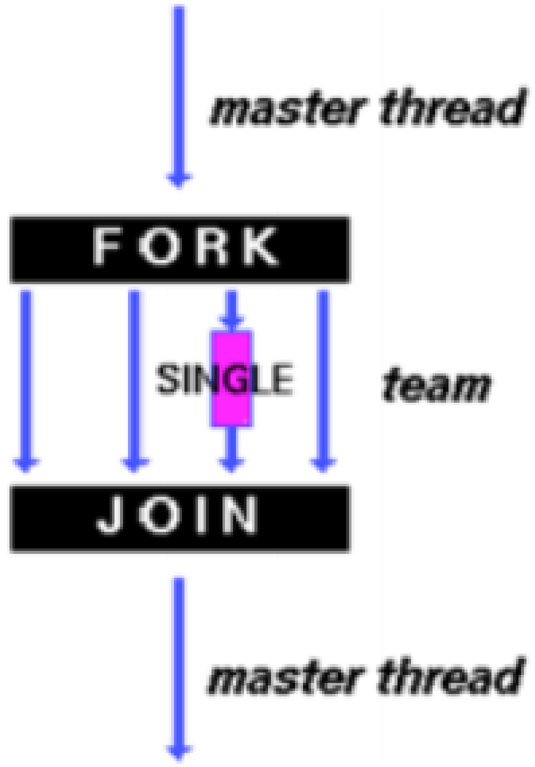
Single and Master

- **Single indicates that only one thread in team will execute the code.**

  #pragma omp single

- **Master indicates that only the master thread will execute the code.**

  #pragma omp master



Critical with Names

- **Critical ensures that only one thread can execute code block at a time.**

```
# pragma omp critical
  global_result += my_result ;
```

- **You can name critical sections, then critical sections with different names can be executed in parallel**

```
# pragma omp critical(name)
  global_result += my_result ;
```

Example 7.16 Using the critical directive for producer-consumer threads

```
#pragma omp parallel sections
{
    #pragma parallel section
    {
        /* producer thread */
        task = produce_task();
        #pragma omp critical ( task_queue)
        {
            insert_into_queue(task);
        }
    }
    #pragma parallel section
    {
        /* consumer thread */
        #pragma omp critical ( task_queue)
        {
            task = extract_from_queue(task);
        }
        consume_task(task);
    }
}
```

- A producer thread generates a task and inserts it into a task-queue.
- The consumer thread extracts tasks from the queue and executes them one at a time.
- Since there is concurrent access to the task-queue, these accesses must be serialized using critical blocks.
- Specifically, the tasks of inserting and extracting from the task-queue must be serialized.
- Note that queue full and queue empty conditions must be explicitly handled here in functions insert_into_queue and extract_from_queue.

7.10.4 Data Handling in OpenMP

Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.
- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

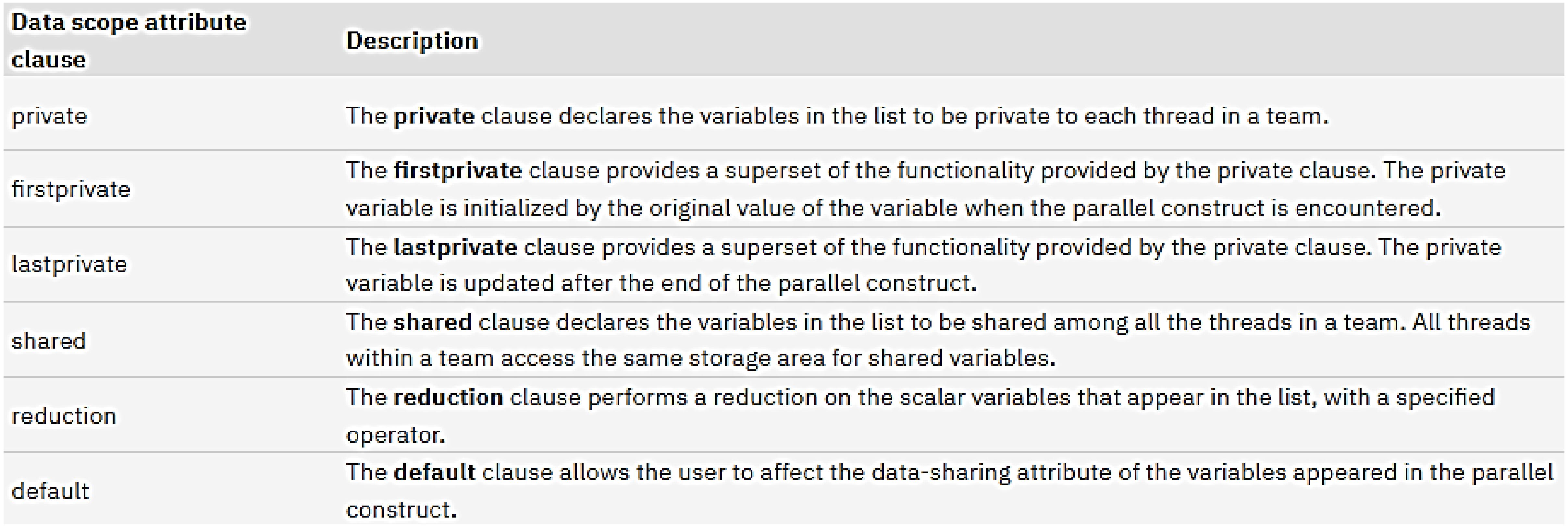
Scope in OpenMP

- A variable that can be accessed by all the threads in the team has shared scope.
- A variable that can only be accessed by a single thread has private scope.
- The default scope for variables declared before a parallel block is shared.

7.10.4 Data Handling in OpenMP

One of the critical factors influencing program performance is the manipulation of data by threads. We have briefly discussed OpenMP support for various data classes such as `private`, `shared`, `firstprivate`, and `lastprivate`. We now examine these in greater detail, with a view to understanding how these classes should be used. We identify the following heuristics to guide the process:

- If a thread initializes and uses a variable (such as loop indices) and no other thread accesses the data, then a local copy of the variable should be made for the thread. Such data should be specified as `private`.
- If a thread repeatedly reads a variable that has been initialized earlier in the program, it is beneficial to make a copy of the variable and inherit the value at the time of thread creation. This way, when a thread is scheduled on the processor, the data can reside at the same processor (in its cache if possible) and accesses will not result in interprocessor communication. Such data should be specified as `firstprivate`.
- If multiple threads manipulate a single piece of data, one must explore ways of breaking these manipulations into local operations followed by a single global operation. For example, if multiple threads keep a count of a certain event, it is beneficial to keep local counts and to subsequently accrue it using a single summation at the end of the parallel block. Such operations are supported by the `reduction` clause.
- If multiple threads manipulate different parts of a large data structure, the programmer



Data scope attribute clause	Description
private	The **private** clause declares the variables in the list to be private to each thread in a team.
firstprivate	The **firstprivate** clause provides a superset of the functionality provided by the private clause. The private variable is initialized by the original value of the variable when the parallel construct is encountered.
lastprivate	The **lastprivate** clause provides a superset of the functionality provided by the private clause. The private variable is updated after the end of the parallel construct.
shared	The **shared** clause declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The **reduction** clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
default	The **default** clause allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct.

Scoping in OpenMP: Dividing variables in *shared* and *private*:

- → *private*-list and *shared*-list on Parallel Region
- → *private*-list and *shared*-list on Worksharing constructs
- → General default is *shared* for Parallel Region.
- → Loop control variables on *for*-constructs are *private*
- → Non-static variables local to Parallel Regions are *private*
- → *private*: A new uninitialized instance is created for each thread
  - → *firstprivate*: Initialization with Master‘s value
  - → *lastprivate*: Value of last loop iteration is written back to Master
- → Static variables are *shared*

The default clause

- Lets the programmer specify the scope of each variable in a block.

```
default(none)
```

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

7.10.5 OpenMP Library Functions

Controlling Number of Threads and Processors

```
#include <omp.h>

void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```

Lecture # 18 – Topics

- Limits on Parallel Performance
- Task Interaction Graph
    - Introduction
    - Comparaison with Task dépendancy graph.

Limits on Parallel Performance

- It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity.
- There is an inherent bound on how fine the granularity of a computation can be. *For example, in the case of multiplying a dense matrix with a vector, there can be no more than* ***(n^2)*** *concurrent tasks.*
- Concurrent tasks may also have to exchange data with other tasks. This results in communication overhead. The tradeoff between the granularity of a decomposition and associated overheads often determines performance bounds.

Task Interaction Graphs

- Subtasks generally exchange data with others in a decomposition.
  - For example, even in the trivial decomposition of the dense matrix-vector product, if the vector is not replicated across all tasks, they will have to communicate elements of the vector.

- The graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a *task interaction graph*.

  - *Task interaction graphs* represent data dependencies,
  - *Task dependency graphs* represent control dependencies.

The pattern of interaction among tasks is captured by what is known as a ***task-interaction graph***. The nodes in a task-interaction graph represent tasks and the edges connect tasks that interact with each other. The nodes and edges of a task-interaction graph can be assigned weights proportional to the amount of computation a task performs and the amount of interaction that occurs along an edge, if this information is known. The edges in a task-interaction graph are usually undirected, but directed edges can be used to indicate the direction of flow of data, if it is unidirectional. The edge-set of a task-interaction graph is usually a superset of the edge-set of the task-dependency graph. In the database query example discussed earlier, the task-interaction graph is the same as the task-dependency graph. We now give an example of a more interesting task-interaction graph that results from the problem of sparse matrix-vector multiplication.

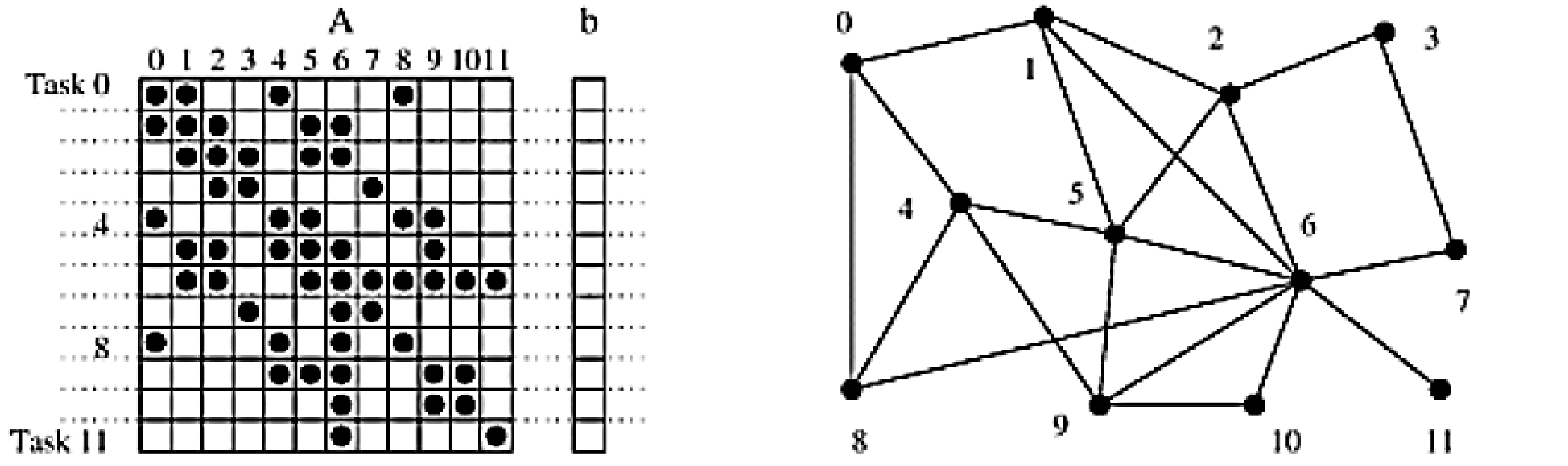
Example 3.3 Sparse matrix-vector multiplication

Consider the problem of computing the product *y* = *Ab* of a sparse *n* x *n* matrix *A* with a dense *n* x 1 vector *b*. A matrix is considered sparse when a significant number of entries in it are zero and the locations of the non-zero entries do not conform to a predefined structure or pattern. Arithmetic operations involving sparse matrices can often be optimized significantly by avoiding computations involving the zeros.

Task Interaction graph- An example

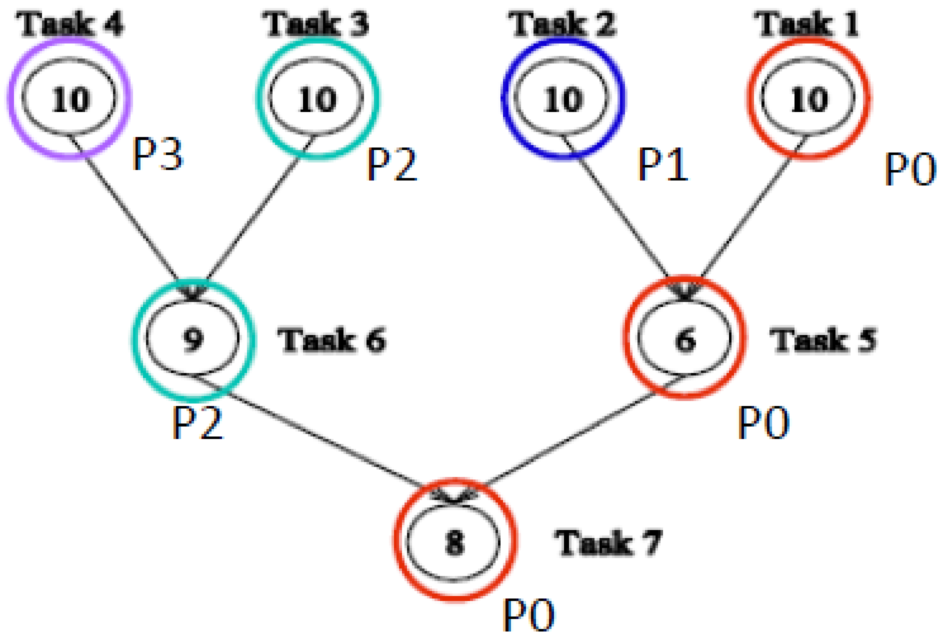
$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

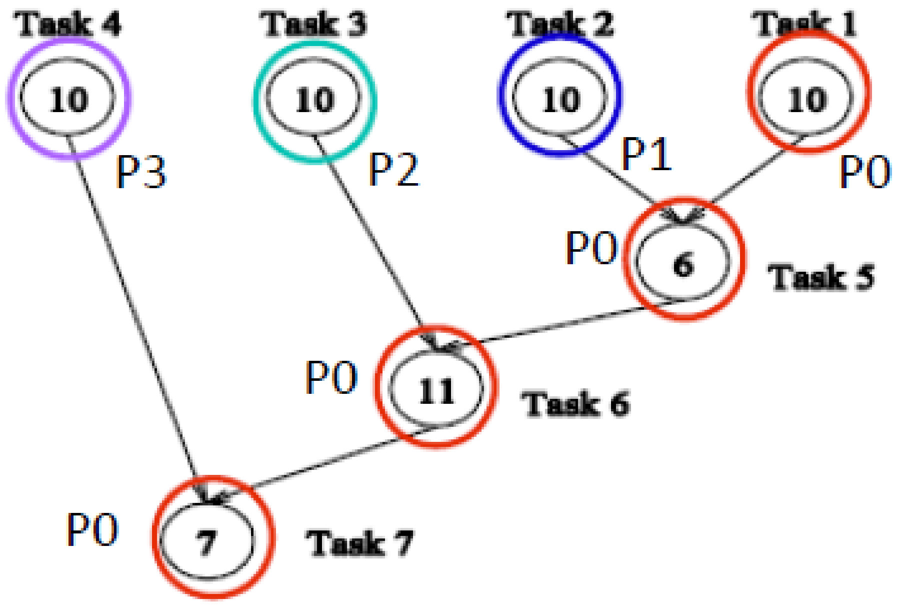
Figure 3.6. A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task *i* computes $\sum_{0 \le j \le 11, A[i,j] \ne 0} A[i,j].b[j]$**.**



(a) (b)

Example: Mapping Database Query to Processes





No two nodes in a level have dependencies, therefore, single level tasks are assigned to different processes.

- 4 processes can be used in total since the max. concurrency is 4.
- Assign all tasks within a level to different processes.