Lab Assignment -1

Basic functions of OpenMP and creating a c program for dot product using OpenMP

# Done By: Arshdeep Singh Bhatia

# Registration number: 19BCB0086

Submitted to: Prof. Balamurugan R



# Task 1

Implement the OpenMP standard includes the following functions and data types using C and include the appropriate declarations of the routines in your source code and explain it.

## omp\_init\_lock

Initializes a simple lock. Needs variable of type omp\_lock\_t.

void omp\_init\_lock (

omp\_lock\_t \*lock

);

## omp\_get\_thread\_num

Returns the thread number of the thread executing within its thread team.

int omp\_get\_thread\_num ();

## omp\_set\_lock

Blocks thread execution until a lock is available.

void omp\_set\_lock(

omp\_lock\_t \*lock

);

## omp\_unset\_lock

Releases a lock

void omp\_unset\_lock(

omp\_lock\_t \*lock

);

## omp\_destroy\_lock

Uninitializes a lock. Works on variable of type omp\_lock\_t that was initialized with omp\_init\_lock.

void omp\_destroy\_lock (

omp\_lock\_t \*lock

);

CODE involving functions 1-5

#include <stdio.h>

#include <omp.h>

omp\_lock\_t my\_lock;

int main()

{

   omp\_init\_lock(&my\_lock);

   #pragma omp parallel num\_threads(4)

   {

      int tid = omp\_get\_thread\_num( );

      int i, j;

      for (i = 0; i < 2; ++i) {

         omp\_set\_lock(&my\_lock);

         printf("Thread %d - starting locked region\n", tid);

         printf("Thread %d - ending locked region\n", tid);

         omp\_unset\_lock(&my\_lock);

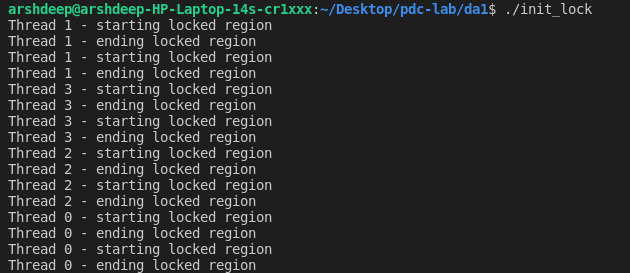
      }

   }

   omp\_destroy\_lock(&my\_lock);

}

Output



## omp\_set\_dynamic

Indicates that the number of threads available in upcoming parallel regions can be adjusted by the run time. A value that indicates if the number of threads available in upcoming parallel regions can be adjusted by the runtime. If nonzero, the runtime can adjust the number of threads, if zero, the runtime won't dynamically adjust the number of threads.

void omp\_set\_dynamic(

int val

);

## omp\_set\_num\_threads

Sets the number of threads in upcoming parallel regions, unless overridden by a num\_threads clause.

void omp\_set\_num\_threads(

int num\_threads

);

## omp\_get\_dynamic

Returns a value that indicates if the number of threads available in upcoming parallel regions can be adjusted by the run time.

int omp\_get\_dynamic();

Code involving functions 6-8

#include <stdio.h>

#include <omp.h>

int main()

{

    omp\_set\_dynamic(9);

    omp\_set\_num\_threads(4);

    printf("%d\n", omp\_get\_dynamic( ));

    #pragma omp parallel

        #pragma omp master

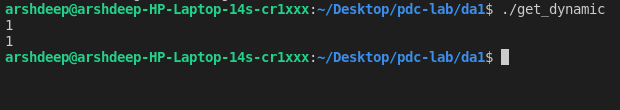
        {

            printf("%d\n", omp\_get\_dynamic( ));

        }

}

Output



## omp\_get\_max\_threads

Returns an integer that is equal to or greater than the number of threads that would be available if a parallel region without num\_threads were defined at that point in the code.

int omp\_get\_max\_threads()

#include <stdio.h>

#include <omp.h>

int main( )

{

    omp\_set\_num\_threads(8);

    printf("%d\n", omp\_get\_max\_threads( ));

    #pragma omp parallel

        #pragma omp master

        {

            printf("%d\n", omp\_get\_max\_threads( ));

        }

    printf("%d\n", omp\_get\_max\_threads( ));

    #pragma omp parallel num\_threads(3)

        #pragma omp master

        {

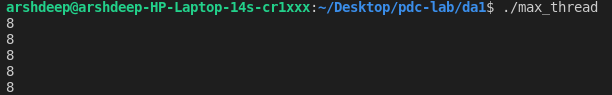
            printf("%d\n", omp\_get\_max\_threads( ));

        }

    printf("%d\n", omp\_get\_max\_threads( ));

}

Output



## omp\_set\_nested

Enables nested parallelism. A nonzero value enables nested parallelism, while zero disables nested parallelism.

void omp\_set\_nested(

int val

);

## omp\_get\_nested

Returns a value that indicates if nested parallelism is enabled. A nonzero value means nested parallelism is enabled.

int omp\_get\_nested( );

CODE using functions 10 and 11

#include <stdio.h>

#include <omp.h>

int main( )

{

    omp\_set\_nested(1);

    omp\_set\_num\_threads(4);

    printf("%d\n", omp\_get\_nested( ));

    #pragma omp parallel

        #pragma omp master

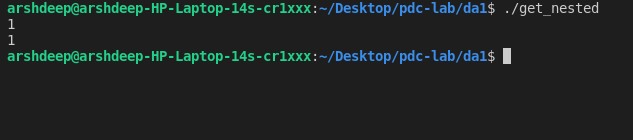
        {

            printf("%d\n", omp\_get\_nested( ));

        }

}

Output



## omp\_get\_num\_procs

Returns the number of processors that are available when the function is called. Basically 8 for my computer.

int omp\_get\_num\_procs();

#include <stdio.h>

#include <omp.h>

int main( )

{

    printf("%d\n", omp\_get\_num\_procs( ));

    #pragma omp parallel

        #pragma omp master

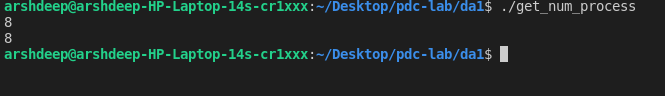
        {

            printf("%d\n", omp\_get\_num\_procs( ));

        }

}

Output



## omp\_get\_num\_threads

Returns the number of threads in the parallel region.

int omp\_get\_num\_threads( );

int main()

{

    omp\_set\_num\_threads(4);

    printf("%d\n", omp\_get\_num\_threads( ));

    #pragma omp parallel

        #pragma omp master

        {

            printf("%d\n", omp\_get\_num\_threads( ));

        }

    printf("%d\n", omp\_get\_num\_threads( ));

    #pragma omp parallel num\_threads(3)

        #pragma omp master

        {

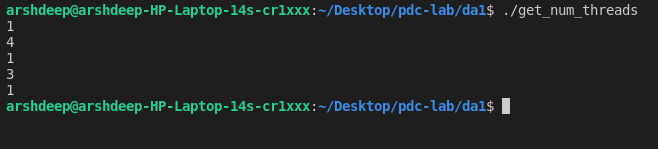
            printf("%d\n", omp\_get\_num\_threads( ));

        }

    printf("%d\n", omp\_get\_num\_threads( ));

}

Output



## omp\_get\_default\_device ()

gets the value of the default device

CODE

#include <stdio.h>

#include <omp.h>

int main( )

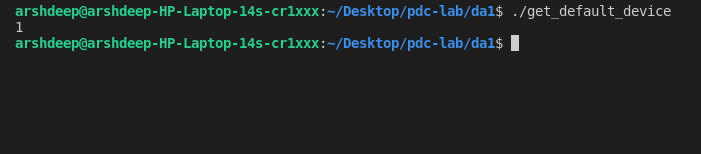
{

    omp\_set\_default\_device(1);

    printf("%d\n", omp\_get\_default\_device( ));

}

OUTPUT



## omp\_get\_wtick

Returns the number of seconds between processor clock ticks.

double omp\_get\_wtick( );

## omp\_get\_wtime

Returns a value in seconds of the time elapsed from some point. Returns a value in seconds of the time elapsed from some arbitrary, but consistent point.

double omp\_get\_wtime( );

CODE with function 15 and 16

#include "omp.h"

#include <stdio.h>

int main() {

    double start = omp\_get\_wtime( );

    double end = omp\_get\_wtime( );

    double wtick = omp\_get\_wtick( );

    printf("start = %.16g\nend = %.16g\ndiff = %.16g\n",

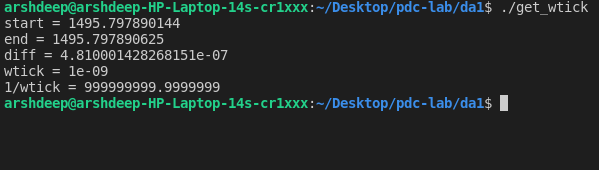
             start, end, end - start);

    printf("wtick = %.16g\n1/wtick = %.16g\n",

             wtick, 1.0 / wtick);

}

Output



## omp\_in\_parallel

Returns nonzero if called from within a parallel region.

int omp\_in\_parallel( );

#include <stdio.h>

#include <omp.h>

int main( )

{

    omp\_set\_num\_threads(4);

    printf("%d\n", omp\_in\_parallel( ));

    #pragma omp parallel

        #pragma omp master

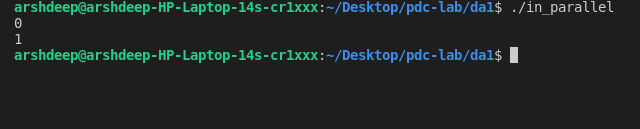
        {

            printf("%d\n", omp\_in\_parallel( ));

        }

}

Output



## omp\_init\_nest\_lock

Initializes a lock.

Parameter used is variable of type omp\_nest\_lock\_t.

void omp\_init\_nest\_lock(

omp\_nest\_lock\_t \*lock

);

## omp\_set\_nest\_lock

Blocks thread execution until a lock is available. Works on variable of type omp\_nest\_lock\_t that was initialized with omp\_init\_nest\_lock

void omp\_set\_nest\_lock(

omp\_nest\_lock\_t \*lock

);

## omp\_unset\_nest\_lock

Releases a nestable lock. Works on variable of type omp\_nest\_lock\_t that was initialized with omp\_init\_nest\_lock, owned by the thread and executing in the function.

void omp\_unset\_nest\_lock(

omp\_nest\_lock\_t \*lock

);

#include <stdio.h>

#include <omp.h>

omp\_nest\_lock\_t my\_lock;

void Test() {

   int tid = omp\_get\_thread\_num( );

   omp\_set\_nest\_lock(&my\_lock);

   printf("Thread %d - starting nested locked region\n", tid);

   printf("Thread %d - ending nested locked region\n", tid);

   omp\_unset\_nest\_lock(&my\_lock);

}

int main() {

   omp\_init\_nest\_lock(&my\_lock);

   #pragma omp parallel num\_threads(4)

   {

      int i, j;

      for (i = 0; i < 5; ++i) {

         omp\_set\_nest\_lock(&my\_lock);

            if (i % 3)

               Test();

            omp\_unset\_nest\_lock(&my\_lock);

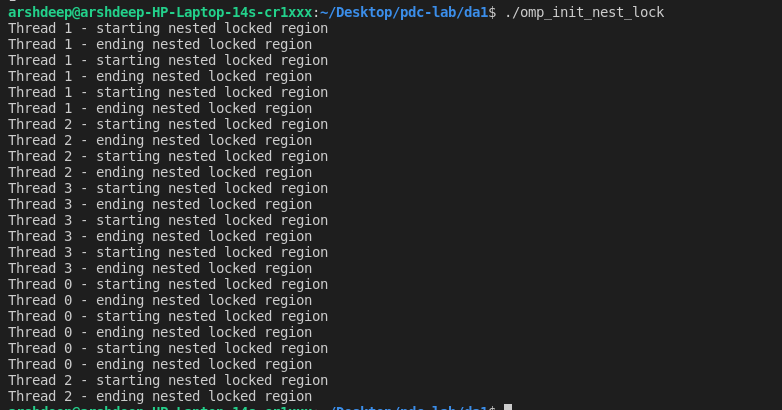
        }

    }

    omp\_destroy\_nest\_lock(&my\_lock);

}

Output



## omp\_test\_lock

Attempts to set a lock but doesn't block thread execution. Zero means failed and nonzero means lock acquired.

int omp\_test\_lock(

omp\_lock\_t \*lock

);

#include <stdio.h>

#include <omp.h>

omp\_lock\_t simple\_lock;

int main() {

    omp\_init\_lock(&simple\_lock);

    #pragma omp parallel num\_threads(4)

    {

        int tid = omp\_get\_thread\_num();

        while (!omp\_test\_lock(&simple\_lock))

            printf("Thread %d - failed to acquire simple\_lock\n",

                     tid);

        printf("Thread %d - acquired simple\_lock\n", tid);

        printf("Thread %d - released simple\_lock\n", tid);

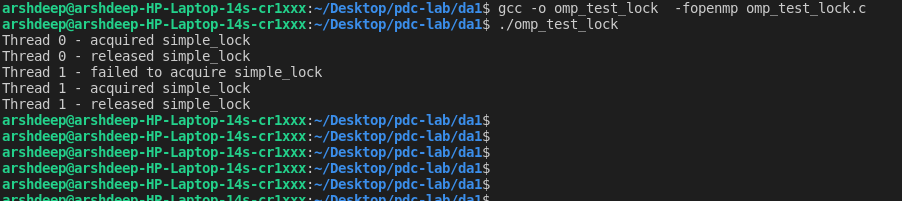
        omp\_unset\_lock(&simple\_lock);

    }

    omp\_destroy\_lock(&simple\_lock);

}

Output



# Task 2

Using OpenMP, Design, develop and run a multi-threaded program to perform dot product and explain it

CODE

#include <omp.h>

#include <stdio.h>

int main()

{

    int i, n, chunk;

    float a[3], b[3], result;

    result = 0.0;

    a[0] = 1;

    a[1] = 2;

    a[2] = 2;

    // vector 1i+2j+2k

    printf("Vector 1 is ");

    for (int i=0;i<3;i++)

    {

        printf("%f ", a[i]);

    }

    b[0] = 3;

    b[1] = 4;

    b[2] = -2;

    // vector 3i+4j-2k

    printf("\nVector 2 is ");

    for (int i=0;i<3;i++)

    {

        printf("%f ", b[i]);

    }

#pragma omp parallel for default(shared) private(i) schedule(static, chunk) reduction(+ : result)

    for (i = 0; i < 3; i++)

        result += (a[i] \* b[i]);

    printf("\nDot product result= %f\n", result);

}

Output

