Lab 2 part 1:

Compilation and output:

Text

Description automatically generated

dot1s.c listing:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FILE: dotprod\_serial.c

\* DESCRIPTION:

\* This is a simple serial program which computes the dot product of two

\* vectors. The threaded version can is dotprod\_mutex.c.

\* SOURCE: Vijay Sonnad, IBM

\* LAST REVISED: 01/29/09 Blaise Barney

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

/\*

The following structure contains the necessary information

to allow the function "dotprod" to access its input data and

place its output so that it can be accessed later.

\*/

**typedef** **struct**

{

**double** \*a;

**double** \*b;

**double** sum;

**int** veclen;

} DOTDATA;

// int VECLEN = 1000000;

DOTDATA dotstr;

/\*

We will use a function (dotprod) to perform the scalar product.

All input to this routine is obtained through a structure of

type DOTDATA and all output from this function is written into

this same structure. While this is unnecessarily restrictive

for a sequential program, it will turn out to be useful when

we modify the program to compute in parallel.

\*/

**void** dotprod()

{

/\* Define and use local variables for convenience \*/

**int** start, end, i;

**double** mysum, \*x, \*y;

start=0;

end = dotstr.veclen;

x = dotstr.a;

y = dotstr.b;

/\*

Perform the dot product and assign result

to the appropriate variable in the structure.

\*/

mysum = 0;

**for** (i=start; i<end ; i++)

{

mysum += (x[i] \* y[i]);

}

dotstr.sum = mysum;

}

/\*

The main program initializes data and calls the dotprd() function.

Finally, it prints the result.

\*/

**int** main (**int** argc, **char** \*argv[])

{

// modify the program to get VECLEN as the argument1

// that is, run the program: ./dots 100000

**int** VECLEN = atoi(argv[1]);

**int** i,len;

**double** \*a, \*b;

/\* Assign storage and initialize values \*/

len = VECLEN;

a = (**double**\*) malloc (len\***sizeof**(**double**));

b = (**double**\*) malloc (len\***sizeof**(**double**));

**for** (i=0; i<len; i++) {

a[i]=1;

b[i]=a[i];

}

dotstr.veclen = len;

dotstr.a = a;

dotstr.b = b;

dotstr.sum=0;

/\* Perform the dotproduct \*/

dotprod ();

/\* Print result and release storage \*/

printf ("Sum = %f \n", dotstr.sum);

free (a);

free (b);

}

Lab 2 part 2:

Compilation and output:

Graphical user interface, text, application

Description automatically generated

dot1m.c listing:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FILE: dotprod\_mutex.c

\* DESCRIPTION:

\* This example program illustrates the use of mutex variables

\* in a threads program. This version was obtained by modifying the

\* serial version of the program (dotprod\_serial.c) which performs a

\* dot product. The main data is made available to all threads through

\* a globally accessible structure. Each thread works on a different

\* part of the data. The main thread waits for all the threads to complete

\* their computations, and then it prints the resulting sum.

\* SOURCE: Vijay Sonnad, IBM

\* LAST REVISED: 01/29/09 Blaise Barney

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

/\*

The following structure contains the necessary information

to allow the function "dotprod" to access its input data and

place its output into the structure. This structure is

unchanged from the sequential version.

\*/

**typedef** **struct** {

**double** \*a;

**double** \*b;

**double** sum;

**int** veclen;

} DOTDATA;

/\* Define globally accessible variables and a mutex \*/

/\* now moved to main \*/

// #define NUMTHRDS 10

// #define VECLEN 100000

// pthread\_t callThd[NUMTHRDS];

pthread\_mutex\_t mutexsum;

DOTDATA dotstr;

/\*

The function dotprod is activated when the thread is created.

As before, all input to this routine is obtained from a structure

of type DOTDATA and all output from this function is written into

this structure. The benefit of this approach is apparent for the

multi-threaded program: when a thread is created we pass a single

argument to the activated function - typically this argument

is a thread number. All the other information required by the

function is accessed from the globally accessible structure.

\*/

**void** \*dotprod(**void** \*arg) {

/\* Define and use local variables for convenience \*/

**int** i, start, end, len ;

**long** offset;

**double** mysum, \*x, \*y;

offset = (**long**)arg;

len = dotstr.veclen;

start = offset\*len;

end = start + len;

x = dotstr.a;

y = dotstr.b;

/\*

Perform the dot product and assign result

to the appropriate variable in the structure.

\*/

mysum = 0;

**for** (i=start; i<end ; i++) {

mysum += (x[i] \* y[i]);

}

/\*

Lock a mutex prior to updating the value in the shared

structure, and unlock it upon updating.

\*/

pthread\_mutex\_lock (&mutexsum);

dotstr.sum += mysum;

printf("Thread %ld did %d to %d: mysum=%f global sum=%f\n",

offset,start,end,mysum,dotstr.sum);

pthread\_mutex\_unlock (&mutexsum);

pthread\_exit((**void**\*) 0);

}

/\*

The main program creates threads which do all the work and then

print out result upon completion. Before creating the threads,

The input data is created. Since all threads update a shared structure, we

need a mutex for mutual exclusion. The main thread needs to wait for

all threads to complete, it waits for each one of the threads. We specify

a thread attribute value that allow the main thread to join with the

threads it creates. Note also that we free up handles when they are

no longer needed.

\*/

**int** main (**int** argc, **char** \*argv[]) {

// modify the program to get NUMTHRDS as the argument1

// and VECLEN as argument2

// that is, run the program: ./dotm 10 100000

//

**int** NUMTHRDS = atoi(argv[1]);

**int** VECLEN = atoi(argv[2]);

pthread\_t callThd[NUMTHRDS];

**long** i;

**double** \*a, \*b;

**void** \*status;

pthread\_attr\_t attr;

/\* Assign storage and initialize values \*/

a = (**double**\*) malloc (NUMTHRDS\*VECLEN\***sizeof**(**double**));

b = (**double**\*) malloc (NUMTHRDS\*VECLEN\***sizeof**(**double**));

**for** (i=0; i<VECLEN\*NUMTHRDS; i++) {

a[i]=1;

b[i]=a[i];

}

dotstr.veclen = VECLEN;

dotstr.a = a;

dotstr.b = b;

dotstr.sum=0;

pthread\_mutex\_init(&mutexsum, **NULL**);

/\* Create threads to perform the dotproduct \*/

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);

**for**(i=0;i<NUMTHRDS;i++) {

/\* Each thread works on a different set of data.

\* The offset is specified by 'i'. The size of

\* the data for each thread is indicated by VECLEN.

\*/

pthread\_create(&callThd[i], &attr, dotprod, (**void** \*)i);

}

pthread\_attr\_destroy(&attr);

/\* Wait on the other threads \*/

**for**(i=0;i<NUMTHRDS;i++) {

pthread\_join(callThd[i], &status);

}

/\* After joining, print out the results and cleanup \*/

printf ("Sum = %f \n", dotstr.sum);

free (a);

free (b);

pthread\_mutex\_destroy(&mutexsum);

pthread\_exit(**NULL**);

Lab 2 part 3:

A picture containing chart

Description automatically generated



A picture containing graphical user interface

Description automatically generated



Analysis:

In terms of speed, parallel run with threads is faster than serial run. The total time used by dot1m.c runs were faster than the dot1s.c run.

Parallel runs require much more from the CPU time than serial run. The runs by dot1m.c were awarded between 112% - 129% of CPU time, while the run by dot1s.c was awarded only 67% of CPU time.

Out of the 5 dot1m.c runs with different amount of threads; the best run would be dot1m.c 2 16000000 because it needed the least amount of CPU time.

Increasing more threads leads to performance decrease, since the CPU time required increases. However, when it comes to total elapsed time which counts overhead time factors, the middle 2 runs, dot1m.c 8 16000000 and dot1m.c 16 2000000, took the least amount of time. Hence, if measuring performance by total elapsed time, balancing the number of threads with the VECLEN value should be ideal.