National University of Computer and Emerging Sciences

Operating System Lab – 09

Lab Manual

Contents

Objective

Previously we have learnt about basics of threads, we have learnt how we create threads and use them in our computations, we learnt that how we can set attributes and pass arguments of/to threads and we also have seen how to join and synchronize threads. In today's lab we will learn some more computations with 'pthread' Library and then we will see basics of OpenMP along with that we will compare OpenMP with Pthread.

More Computations with 'pthread'

Computation of e^x with Taylor Sequence

Taylor Sequence:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^r}{r!} + ... \qquad (\text{all } x)$$

Code:

```
#include<math.h>
#include<pthread.h>
#include<stdlib.h>

long double x,fact[150], pwr[150],s[1];
int i,term;

void *Power(void *temp) {
        int k;
        for(k=0;k<150;k++) {
        pwr[k] = pow(x,k);
                //printf("%.2Lf\n",pwr[k]);
        }
        return pwr;
}
void *Fact(void *temp) {
        long double f;
        int j;
        fact[0] = 1.0;
        for(term=1;term<150;term++) {
            f = 1.0;
            for(j=term;j>0;j--)
                f = f * j;
            fact[term] = f;
            //printf("%.2Lf\n",fact[term]);
        }
        return fact;
}
```

```
void *Exp(void *temp) {
        int t;
        s[0] = 0;
        for(t=0;t<150;t++)
        s[0] = s[0] +  (pwr[t] / fact[t]);

        return s;
}

int main(void) {
        pthread_t thread1,thread2,thread3;
        long double **sum;
        printf("Exponential [PROMPT] Enter the value of x (between 0 to 100) (for calculating exp(x)):
");
        scanf("%Lf",&x);

        printf("\nExponential [INFO] Threads creating.....\n");
        pthread_create(&thread1,NULL,Power,NULL); //calling power function
        pthread_create(&thread2,NULL,Fact, NULL); //calling factorial function
        printf("Exponential [INFO] Threads created\n");

        pthread_join(thread1,NULL);
        pthread_join(thread2,NULL);
        printf("Exponential [INFO] Master thread and terminated threads are joining\n");
        printf("Exponential [INFO] Result collected in Master thread\n");

        pthread_create(&thread3,NULL,Exp,NULL);
        pthread_join(thread3,sum);
        printf("\neXPONENTIAL [INFO] Value of exp(%.2Lf) is : %Lf\n\n",x,s[0]);

        exit(1);
}
```

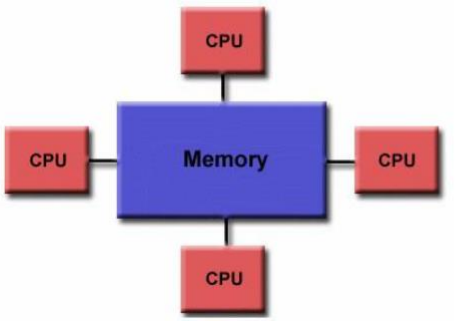
Introduction to OpenMP

OpenMP Is: An Application Program Interface (API) that may be used to explicitly direct ***multithreaded, shared memory*** parallelism. Comprised of three primary API components:

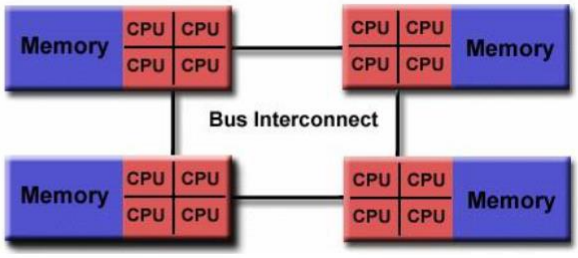
- Compiler Directives
- Runtime Library Routines
- Environment Variables

Shared Memory Model:

OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.



Uniform Memory Access



Non-Uniform Memory Access

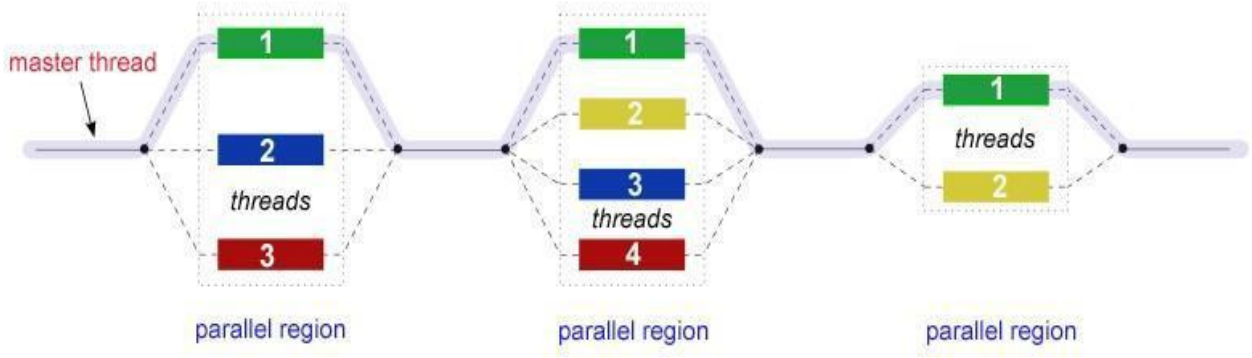
Thread Based Parallelism:

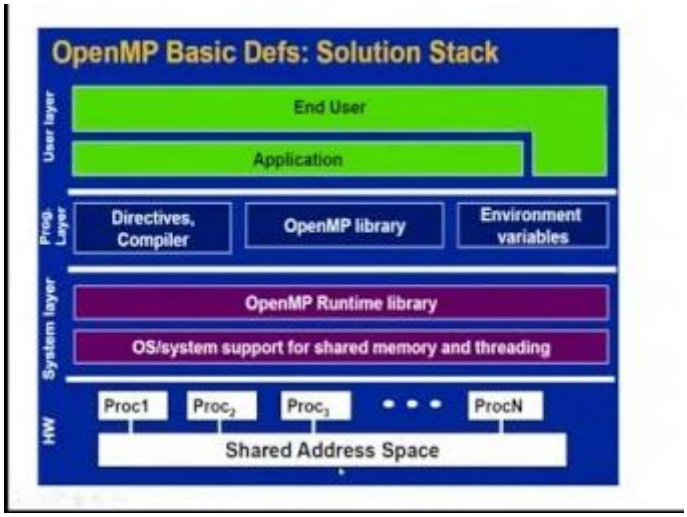
- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
- Threads exist within the resources of a single process. Without the process, they cease to exist.
- Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

Explicit Parallelism:

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- 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.

Fork - Join Model:





Three Components:

The OpenMP API is comprised of three distinct components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

The application developer decides how to employ these components. In the simplest case, only a few of them are needed.

Implementations differ in their support of all API components. For example, an implementation may state that it supports nested parallelism, but the API makes it clear that may be limited to a single thread - the master thread. Not exactly what the developer might expect?

Compiler Directives:

Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag, as discussed in the Compiling section later.

OpenMP compiler directives are used for various purposes:

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

Compiler directives have the following syntax:

sentinel directive-name [clause, ...]

For example: for C/C++

#pragma omp parallel default(shared) private(beta,pi)

Run-time Library Routines:

The OpenMP API includes an ever-growing number of run-time library routines.
These routines are used for a variety of purposes:

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
- Setting and querying the dynamic threads feature
- Querying if in a parallel region, and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks and nested locks
- Querying wall clock time and resolution

For C/C++, all of the run-time library routines are actual subroutines.

```
#include <omp.h>
int omp_get_num_threads(void)
```

- Max number of threads

  omp_get_max_threads()
- Number of processors

  omp_get_num_procs()
- Number of threads (inside a parallel region)

  omp_get_num_threads()
- Get thread ID

  omp_get_thread_num()

Environment Variables:

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
- These environment variables can be used to control such things as:
    - Setting the number of threads
    - Specifying how loop iterations are divided
    - Binding threads to processors
    - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
    - Enabling/disabling dynamic threads
    - Setting thread stack size
    - Setting thread wait policy

Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which shell you use. For example:

```
export OMP_NUM_THREADS=8
```

Control the Number of Threads

- Parallel region

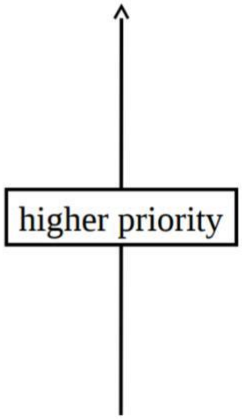
  #pragma omp parallel **num_threads**(*integer*)

- Run-time function

  omp_set_num_threads()

- Environment variable

  export OMP_NUM_THREADS=n



Code Structure

```
#include<omp.h>
int main () {

int var1, var2, var3;
  Serial code
        .
        .
        .
Beginning of parallel region. Fork a team of threads.
   Specify variable scoping
   #pragma omp parallel private(var1, var2) shared(var3)
      {
      Parallel region executed by all threads          .
      Other OpenMP directives
      Run-time Library calls
      All threads join master thread and disband
      }
Resume Serial Code
     .
     .
     .
}
```

Example 1: Table Computation

Below code prints table using OpenMP for loop Parallelization

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

int main() {
          int num;
          int i;
          printf("Table [PROMPT] Enter Your Number: "); scanf("%d",&num);

          #pragma omp parallel num_threads(10)
          #pragma omp for
          for(i=0;i<10000;i++) {
                    printf("Table [INFO] Thread ID: %d | %d X %d = %d \n", omp_get_thread_num(), i, num,
i*num );
          }

          return 0;
}
```

Step 1: Copy this code into a .c File.
Step 2: Compile the using the following command

```
gcc -o table table.c -fopenmp
```

Step 3 (optional) set environmental variables
Step 4: Run the code and observe the processor using system monitor.

Multithreaded program that prints “hello world”

```
#include<omp.h>

void main() {

#pragma omp parallel

{

int ID = 0;

printf(“ hello(%d) ”, ID);

printf(“ world(%d) \n”, ID);

}

}
```

```
#include <iostream>
#include <omp.h>
using namespace std;
int main () {
    int nthreads;
    int tid;
#pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        cout << "Hello World from thread = " << tid << endl;
        if (tid == 0){
            nthreads = omp_get_num_threads();
            cout << "Number of threads = " << nthreads << endl;
        }
    }
}
```

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

int main(int argc, char** argv){
   int partial_Sum, total_Sum;

   #pragma omp parallel private(partial_Sum) shared(total_Sum)
   {
      partial_Sum = 0;
      total_Sum = 0;

      #pragma omp for
      {
         for(int i = 1; i <= 1000; i++){
            partial_Sum += i;
         }
      }

      //Create thread safe region.
      #pragma omp critical
      {
            //add each threads partial sum to the total sum
            total_Sum += partial_Sum;
      }
   }
   printf("Total Sum: %d\n", total_Sum);
   return 0;
}
```

we want to add all number from 1 to 1000, we will initialize out loop at one and end at 1000. Lets first create variables `partial_Sum` and `total_Sum` to hold each thread's partial summation and to hold the total sum of all threads respectively. Next let's begin our parallel section with `pragma omp parallel` . We will also set `partial_Sum` to be a private variable and `total_Sum` to be a shared variable. We shall initialize each variable in the parallel section.

Now we must join our threads. To do this we must use a critical directive to create a thread safe section of code. We do this with `#pragma omp critical` directive. Lastly we add partial sum to total sum and print out the result outside the parallel section of code.

Example 2: Setting Runtime Variables with For Loops

Below code will demonstrate how we can set runtime library routines with OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main (int argc, char *argv[]) {

int i, tid, nthreads, n = 10, N = 100000000;
double *A, *B, tResult, fResult;
time_t start, stop;
clock_t ticks;
long count;
A = (double *) malloc(N*sizeof(double));
B = (double *) malloc(N*sizeof(double));
for (i=0; i<N; i++) {
        A[i] = (double)(i+1);
        B[i] = (double)(i+1);
      }
    time(&start);
    //this block use single process
for (i=0; i<N; i++)
    {
            fResult = fResult + A[i] + B[i];
     }
    //begin of parallel section
    #pragma omp parallel private(tid, i,tResult) shared(n,A,B,fResult)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
          }
    #pragma omp for schedule (static, n)
        for (i=0; i < N; i++) {
            tResult = tResult + A[i] + B[i];
}
    #pragma omp for nowait
        for (i=0; i < n; i++)
        {
           printf("Thread %d does iteration %d\n", tid, i);
}
    #pragma omp critical
        fResult = fResult + tResult;
}
    //end of parallel section
time(&stop);
printf("%f\n",fResult);
printf("Finished in about %.0f seconds. \n", difftime(stop, start));
exit(0);
}
```

Lab Activity

- Execute and understand the Example 2 given in the manual.
- Write a multithreaded program that prints “hello world”.
- Write a multithreaded program where each thread prints “hello world”.
- Matrix Computations are small and independent of each other, each cell in matrix can be computed concurrently without having conflicts.
    - Write a program that calculates the addition of two matrices.
    - Write a program that calculates the multiplication of two matrices.
- Convert the example of Taylor Series into OpenMP.

References:

https://www.codeproject.com/Articles/60176/A-Beginner-s-Primer-to-OpenMP
http://www.openmp.org/resources/
https://idre.ucla.edu/sites/default/files/intro-openmp-2013-02-11.pdf