BACH KHOA UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE & ENGINEERING

Course: Parallel Processing
Lab #1 - Introduction & Multithreads

Minh Thanh Chung

January 23, 2018

Goal: This lab helps student to get familiar with experimental environment on the server - SuperNode-XP. Then, student can review the notation of Thread in Operating System Course and program the multithread program.

Content: Section 1 introduces how to access the server to do examples and exercises. The next Sections show example to practice with multithread program using POSIX Threads and OpenMP.

Result: After this Lab, student can understand and write a program with multithread in parallel.

Contents

1   Introduction

1.1   Contents and Grading

1.2   How to access the server of this course

In this course, each student has an account to access the server and do examples or exercises on it. However, if you want to do the Lab on your laptop, you can set up your own environment with Virtual Machine. Because all of Labs and Assignment will be tested on Linux OS.

For Windows: a simple tool that you can use is Putty (`https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html`). Information of gateway to access from outside networks:
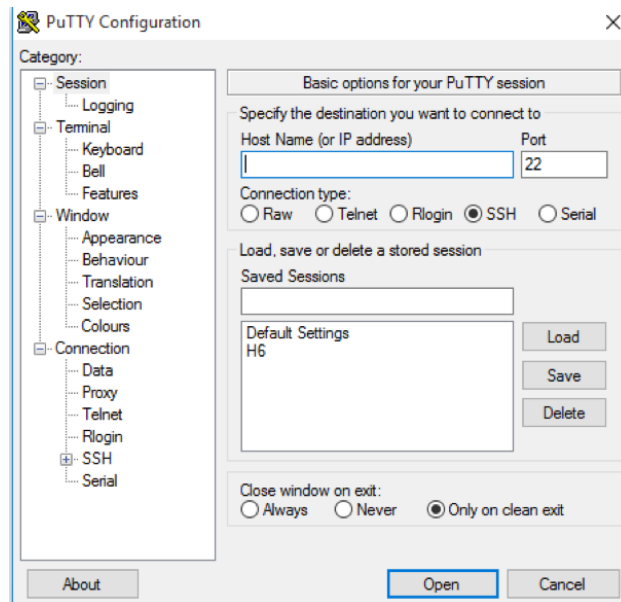

Figure 1: GUI of Putty

- Hostname: **hpcc.hcmut.edu.vn**

- Port: 22

After that, click open. Then, you need an account and password to access the gateway (this is a public account):

- Username: **std**

- Password: **bkhpcc**

At this step, you are just in the gateway, then, you need to access your machine by **ssh** protocol.

```
$ ssh username@IP_address
```

For Linux, you do not need to use other tools, you can access by **ssh** protocol.

```
$ ssh std@hpcc.hcmut.edu.vn

// Then, you continuously ssh to the machine with the given IP address
$ ssh username@IP_address
```

```
5    // For example, you are provided the IP of machine: 10.1.6.1 and user name is 51100123
     // Step 1: ssh to the gateway
     $ ssh std@hpcc.hcmut.edu.vn
     // Step 2: ssh to the machine
10   $ ssh 51100123@10.1.6.1
```

1.3   How to transfer data between host and server

For Windows, you also use a tool to copy data between server and host, WinSCP (`https://winscp.net/eng/download.php`). Information for data transfer (with the window on the left hand side):
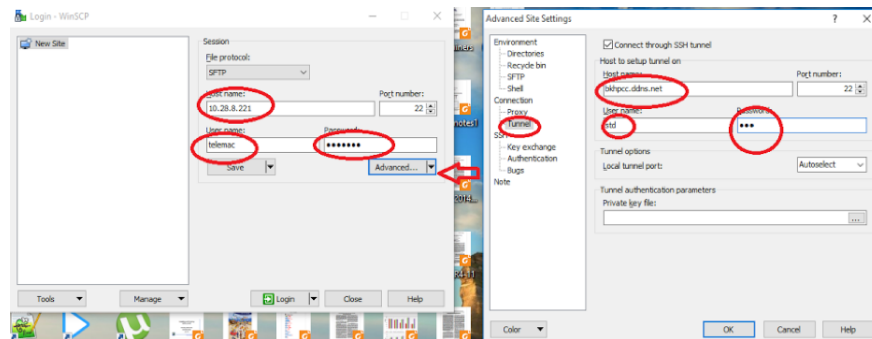

Figure 2: Data transfer between host and server by WinSCP

- Hostname: **IP_address** (of your machine)

- Username: **user_name**

- Password: **pass**

However, to transfer the data, you need pass over the gateway, therefore, in the tab Advanced you need to fill some info such as (note, choose Tunnel at Connection Tab):

- Hostname: **hpcc.hcmut.edu.vn** (address of gateway)

- Username: **std**

- Password: **bkhpcc**

For Linux, you also can copy data by command line:

```
     // Firstly, when you have an account and IP of your machine, you need to create a tunnel to
     // your machine through the gateway. 1234 is the number of port of the tunnel, you can change
     // this number
     $ ssh -L 1234:10.1.6.1:22 std@hpcc.hcmut.edu.vn cat -
5
     // Secondly, you can copy file from local host to the remote machine. At this time, local host is the
     $ scp -P 1234 51100123@127.0.0.1:/home/1510180/a.txt /home/
```

1.4   Set up Virtual Machine on your laptop [Optional]

2   Review

2.1   POSIX Threads - Linux

What is Pthreads?
Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.

Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library - though this library may be part of another library, such as libc, in some implementations.
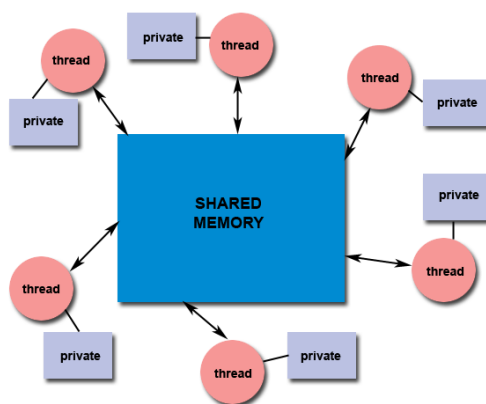

Figure 3: Shared Memory Model

All threads have access to the same global, shared memory. Threads also have their own private data. Programmers are responsible for synchronizing access (protecting) globally shared data.

2.2   Examples

Compiling Threaded Programs: several examples of compile commands used for pthreads codes are listed in the table below.

Table 1: Command lines for compiling Threaded programs

Compiler / Platform	Compiler Command	Description
INTEL Linux	icc -pthread	C
	icpc -pthread	C++
PGI Linux	pgcc -lpthread	C
	pgCC -lpthread	C++
GNULinux, Blue Gene	gcc -pthread	GNU C
	g++ -pthread	GNU C++

Example1: Pthread Creation and Termination

This simple example code creates 5 threads with the pthread create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread exit().

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 10

// user-defined functions
void * user_def_func(void *threadID){
    long TID;
    TID = (long) threadID;
    printf("Hello World! from thread #%ld\n", TID);
    pthread_exit(NULL);
}

int main(int argc, char *argv){
    pthread_t threads[NUM_THREADS];
    int create_flag;
    long i;
    for(i = 0; i < NUM_THREADS; i++){
        printf("In main: creating thread %ld\n", i);
        create_flag = pthread_create(&threads[i], NULL, user_def_func, (void *)i);
        if(create_flag){
            printf("ERROR: return code from pthread_create() is %d\n", create_flag);
            exit(-1);
        }
    }

    // free thread
    pthread_exit(NULL);
    return 0;
}
```

Example2: Thread Argument Passing
This code fragment demonstrates how to pass a simple integer to each thread. The calling thread uses a unique data structure for each thread, insuring that each thread's argument remains intact throughout the program.

```c
...
/* Thread Argument Passing */
// case-study 1
long taskids[NUM_THREADS];

// case-study 2
...

int main (int argc, char *argv[]){
    pthread_t threads[NUM_THREADS];
    int creation_flag;
    long i;
    for(i = 0; i < NUM_THREADS; i++){
        // pass arguments
        taskids[i] = i;
        printf("In main: creating thread %ld\n", i);
```

```
               creation_flag = pthread_create(&threads[i], NULL, user_def_func, (void *)taskids[i]);
                   ...
20       }


       ...
}
```

Question: how to setup/pass multiple arguments via a structure?

Example3: A joinable state for portability purposes

Demonstrates how to explicitly create pthreads in a joinable state for portability purposes. Also shows how to use the *pthread_exit* status parameter.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
5
// Define CONSTANTS
#define NUM_THREADS 4
#define NUM_LOOPS 1000000

10  // user-defined function
void *user_def_func(void *threadID){
      long TID;
      TID = (long) threadID;
      int i;
15      double result = 0.0;
      printf("Thread %ld starting...\n", TID);
      for(i = 0; i < NUM_LOOPS; i++){
            result = result + sin(i) * tan(i);
      }
20
      printf("Thread %ld done. Result = %e\n", TID, result);
      pthread_exit((void*) threadID);
}

25  int main (int argc, char *argv[]){
      pthread_t threads[NUM_THREADS];
      pthread_attr_t attr; // attribute of threads
      int creation_flag, join_flag;
      long i;
30      void *status; // status of threads

      /* Initialize and set thread detached atribute */
      pthread_attr_init(&attr);
      pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
35      for(i = 0; i < NUM_THREADS; i++){
            printf("In main: creating thread %ld\n", i);
            creation_flag = pthread_create(&threads[i], &attr, user_def_func, (void *)i);
            if (creation_flag){
                  printf("ERROR: return code from pthread_create() is %d\n", creation_flag);
40                  exit(-1);
            }
      }
```

```
     /* Free attribute and wait for the other threads */
          pthread_attr_destroy(&attr);
45        for(i = 0; i < NUM_THREADS; i++){
               join_flag = pthread_join(threads[i], &status);
               if (join_flag){
                    printf("ERROR: return code from pthread_join() is %d\n", join_flag);
                    exit(-1);
50             }
               printf("Main: completed join with thread %ld having a status of %ld\n", i, (long)status);
          }

          printf("Main: program completed. Exiting.\n");
55        pthread_exit(NULL);
          return 0;
     }
```

Example4: Race condition
This example uses a mutex variable to protect the global sum while each thread updates it. Race condition is an important problem in parallel programming.

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

5    /* Define global data where everyone can see them */
     #define NUMTHRDS 8
     #define VECLEN 100000
     pthread_mutex_t mutexsum;
     int *a, *b;
10   long sum=0.0;

     void *dotprod(void *arg)
     {
     /* Each thread works on a different set of data.
15    * The offset is specified by the arg parameter. The size of
      * the data for each thread is indicated by VECLEN.
      */
          int i, start, end, offset, len;
          long tid;
20        tid = (long)arg;
          offset = tid;
          len = VECLEN;
          start = offset*len;
          end = start + len;

25
     /* Perform my section of the dot product */
          printf("thread: %ld starting. start=%d end=%d\n",tid,start,end-1);
          for (i=start; i<end ; i++) {
               pthread_mutex_lock(&mutexsum);
30             sum += (a[i] * b[i]);
               pthread_mutex_unlock(&mutexsum);
          }
          printf("thread: %ld done. Global sum now is=%li\n",tid,sum);
          pthread_exit((void*) 0);
```

```
35  }

    int main (int argc, char *argv[])
    {
         long i;
40       void *status;
         pthread_t threads[NUMTHRDS];
         pthread_attr_t attr;
         /* Assign storage and initialize values */
         a = (int*) malloc (NUMTHRDS*VECLEN*sizeof(int));
45       b = (int*) malloc (NUMTHRDS*VECLEN*sizeof(int));
         for (i=0; i<VECLEN*NUMTHRDS; i++)
             a[i]=b[i]=1;


         /* Initialize mutex variable */
50       pthread_mutex_init(&mutexsum, NULL);
    /* Create threads as joinable, each of which will execute the dot product
     * routine. Their offset into the global vectors is specified by passing
     * the "i" argument in pthread_create().
     */
55       pthread_attr_init(&attr);
         pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
         for (i=0;i<NUMTHRDS;i++)
             pthread_create(&threads[i], &attr, dotprod, (void *)i);
         pthread_attr_destroy(&attr);
60
    /* Wait on the other threads for final result */
         for(i=0;i<NUMTHRDS;i++) {
             pthread_join(threads[i], &status);
         }
65
    /* After joining, print out the results and cleanup */
    printf ("Final Global Sum=%li\n",sum);
    free (a);
    free (b);
70
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
    }
```

3   Multithread Programming with OpenMP

3.1   Motivation

What is OpenMP?

- 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

Goals of OpenMP

- Standardization

- Lean and Mean

- Ease of Use

- Portability

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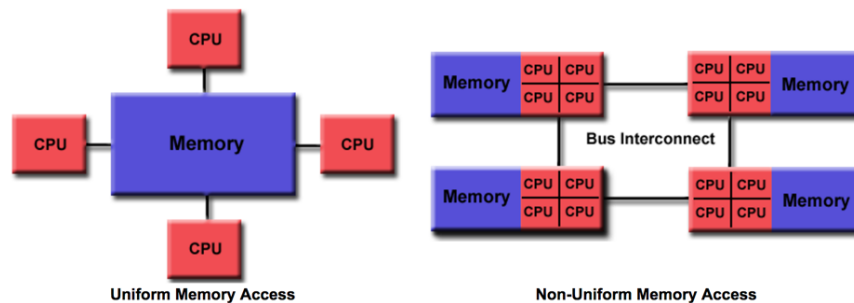

Figure 4: Shared Memory Model for OpenMP

3.2   Examples

Example1: Simple "Hello World" program. Every thread executes all code enclosed in the parallel region. OpenMP library routines are used to obtain thread identifiers and total number of threads.

```
#include <omp.h>

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

  int nthreads, tid;

  /* Fork a team of threads with each thread having a private tid variable */
  #pragma omp parallel private(tid)
   {
```

```
10      /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

15      /* Only master thread does this */
        if (tid == 0)
         {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
20        }


        }  /* All threads join master thread and terminate */
      }
```

Example2: Work-Sharing Constructs - DO / for Directive. The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor

```c
#include <omp.h>

/* Define some values */
#define N 1000
5  #define CHUNKSIZE 100
#define OMP_NUM_THREADS 10
#define MAX_THREADS 48

/* Global variables */
10 int count[MAX_THREADS];
int main(int argc, char **argv){
int i, chunk;
float a[N], b[N], c[N];

15 /* Some initializations */
for(i = 0; i < N; i++){
      a[i] = b[i] = i * 1.0; // values = i with float type
}

20 //for(i = 0; i < OMP_NUM_THREADS; i++){
// count[i] = 0;
//}
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
25 {
      omp_set_num_threads(OMP_NUM_THREADS);
      #pragma omp for schedule(dynamic,chunk) nowait
      for(i = 0; i < N; i++){
            int tid = omp_get_thread_num();
30          printf("Iter %d running from thread %d\n", i, tid);
            c[i] = a[i] + b[i];
            // Increase count[tid]
            count[tid]++
            }
35 }
```

```
     /* Validation */
     printf("Vector c: \n");
     for(i = 0; i < 10; i++){
40       printf("%f ", c[i]);
     }
     printf("...\n");
     /* Statistic */
     //   printf("Num of iter with thread:\n");
45   //   for(i = 0; i < MAX_THREADS; i++){
     //       if(count[i] != 0)
     //       printf("\tThread %d run %d iter.\n", i, count[i]);
     //   }

50   return 0;
     }
```

Example3: Work-Sharing Constructs - SECTIONS Directive

```
     #include <omp.h>

     /* Define some values */
     #define N 1000
5    #define CHUNKSIZE 100
     #define OMP_NUM_THREADS 12
     #define MAX_THREADS 48

     /* Global variables */
10   int count[MAX_THREADS];
     int main(int argc, char **argv){
     int i, chunk;
     float a[N], b[N], c[N], d[N];

15   /* Some initializations */
     for(i = 0; i < N; i++){
         a[i] = i * 1.0;
         b[i] = i + 2.0;
     }
20   for(i = 0; i < OMP_NUM_THREADS; i++){
         count[i] = 0;
     }

     chunk = CHUNKSIZE;
25   #pragma omp parallel shared(a,b,c,d) private(i)
     {
         omp_set_num_threads(OMP_NUM_THREADS);
         #pragma omp sections nowait
     {
30   #pragma omp section
     for(i = 0; i < N; i++){
         int tid_s1 = omp_get_thread_num();
         printf("\tIter %d running from thread %d\n", i, tid_s1);
         c[i] = a[i] + b[i];
35       // Increase count
```

```
            count[tid_s1]++;
        }

        #pragma omp section
40      for(i = 0; i < N; i++){
            int tid_s2 = omp_get_thread_num();
            printf("\tIter %d running from thread %d\n", i, tid_s2);
            d[i] = a[i] * b[i];
            // Increase count
45          count[tid_s2]++;
            }
            }
    }
    /* Validation */
50  printf("Vector c: \n\t");
    for(i = 0; i < 10; i++){
        printf("%f ", c[i]);
    }
    printf("...\n");
55  printf("Vector d: \n\t");
    for(i = 0; i < 10; i++){
        printf("%f ", d[i]);
    }
    printf("...\n");
60
    /* Statistic */
    printf("Num of iter with thread:\n");
    for(i = 0; i < MAX_THREADS; i++){
        if (count[i] != 0)
65          printf("\tThread %d run %d iter.\n", i, count[i]);
    }

    return 0;
    }
```

Example4: THREADPRIVATE Directive The THREADPRIVATE directive is used to make global file scope variables (C/C++) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions.

```
#include <omp.h>
/* Define some values */
#define N 1000
#define CHUNKSIZE 10
5   #define MAX_THREADS 48
#define NUM_THREADS 4
/* Global variables */
int count[MAX_THREADS];
int a, b, i, tid;
10  float x;
#pragma omp threadprivate(a, x)
int main(int argc, char **argv){
    /* Explicitly turn off dynamic threads */
    omp_set_dynamic(0);
15      omp_set_num_threads(NUM_THREADS);
```

```
        printf("1st Parallel Region:\n");
        #pragma omp parallel private(b,tid)
        {
20          tid = omp_get_thread_num();
            25 a = tid;
            b = tid;
            x = 1.1 * tid + 1.0;
            printf("Thread %d: a, b, x = %d, %d, %f\n", tid, a, b, x);
25      }

        printf("*************************************\n");
        printf("Master thread doing serial work here\n");
        printf("*************************************\n");
30      printf("2nd Parallel Region:\n");
        #pragma omp parallel private(tid)
        {
            tid = omp_get_thread_num();
            printf("Thread %d: a, b, x = %d, %d, %f\n", tid, a, b, x);
35      }
        return 0;
}
```

4 Exercises

1. Matrix multiplication with Pthread: implement a parallel version for the given source code with POSIX Thread. Student need to complete //TODO part in the source code. After you finish, lets run the program with matrix sizes: 10, 100, 1000, 10000, 20000 (at least 10000) ... and record the execution time with the command:

```
// For example:
$ time ./mul_mat_pthread_output 1000 1
```

Finally, you plot a graph of performance between Serial Version (already provided in graph.py) and Pthread Version, as Figure 5. In the source code, if you want to modify some variables or data type, it is ok. Note: you can plot the graph on your machine by python. Please search Google to setup Python and plot the graph (Matplotlib library is recommended).

2. OpenMP: pi will be computed by creating a Riemann Integral (http://mathworld.wolfram.com/RiemannIntegral.html) over half a circle, as Figure 6. As the area of a circle with a radius of 1 is equal to , this integral will yield /2. The algorithm implemented is:

- Create an array rect containing the indices 0 to numsteps

- Create an array midPt that contains the middle points of all the rectangles

- Create an array area that contains the area of all the rectangles

- Sum over area and multiply by 2

The code is already prepared in the files pi simple.cpp and pi simple.h.

3. OpenMP: Matrix multiplication is a standard problem in HPC. This computation is exemplified in the Basic Linear Algebra Subroutine (BLAS) function SGEMM. Many libraries contain highly optimized code to execute this problem. In this exercise we define 3 matrices A, B and C of dimension N x N. All elements of matrix A are equal to 1, and all values in B are set to 2. The resulting matrix C should therefore consist of elements equal to 1*2*N.

4. OpenMP: Cholesky Decomposition Algorithm (Bonus) A standard problem in HPC is solving a system of linear equations. What values do you need for a, b, c and d to fulfill these equations?

```
2a + b + 2c + 5d = 24
a + 3b + c + 4d = 15
2a + b + 4c + 7d = 28
5a + 4b + 7c + 3d = -21
```

One solution is the so-called matrix decomposition. In many cases these problems lead to a symmetric (and positive definite) matrix, which can be efficiently decomposed with the Cholesky Decomposition algorithm. Note: The parallel versions of this Cholesky implementation do not scale very well with the number of CPUs.

Note: Student just need to change the given source code provided. All of exercises have the .py file to plot the graph for evaluating performance among scales and problem sizes, so you need to record the results and plot the graph. The number of threads as well as problem sizes is declared in .py files.

5   Submission

5.1   Source code

Requirement: you have to code the program followed by the coding style. Reference: https://www.gnu.org/prep/standards/ht node/Writing-C.html. After you finish the exercise, please compress all of source codes and figure into: lab1_MSSV.zip. The section for submitting the lab will be opened on Sakai. Deadline: 5:00pm, Tue - 30-01-18