Bonus Assignment 1

Ameet Deshpande (CS15B001)

August 28, 2017

1 Concept of threadpool

In the assignments about *pthreads* we were explicitly creating and joining all the threads that were being used in the program. Though this was making the program faster, the programmer had full control over what thread is being used and when the join is happening, etc. This is unnecessary in most cases, and programmers might prefer an easier solution to the same. Enter *threadpool*. Threadpool, as the name goes, is a pool of threads that are constantly waiting to perform some job. When a task is to be performed, the threadpool is checked to see if there are any threads that are not assigned to some other task. If so, it is chosen and given the work. A *thread_pool_manager* is usually implemented to handle this.A typical way of implementing threadpool would be to create a structure for a thread and store an array of structures in another structure called *thread_pool_manager*. There are also implementations available which abstract out these nitty-gritties and provide us with interfaces to deal with. One such CPP library exists, which will be used for the code examples.

2 Why Threadpool

Though threads already give performance improvements, threadpools are very useful to attain concurrency. Firstly, they avoid the overheads that were incurred in thread creation and deletion as they are constantly waiting for a job to perform. That being said, one disadvantage is that the number of threads have to be given initially. If the number of threads is a lot, there will be waste of memory and additional overheads incurred at the time of context switch.
Keeping these in mind, it can be seen that *threadpool* is ideal for cases when there are lots of small similar tasks to be performed and that the max number of these that can parallely run is decided before hand. That brings us to the Matrix Multiplication Problem. This problem has the above properties which make it an eligible candidate for the same.

3 Using Threadpool

Unlike the second Assignment, since we have threads which exist in the pool and are being used from there, we can invoke one thread for each cell of the final matrix. We can set the initial threads to something like3 or 4 if the number of cores in the system are 4. The code logic is to use one thread for each cell in the answer matrix denoted by C and keep accessing the thread pool. The command $pp.push(function\_name, functor)$ pushes the thread pool request in and that process is executed once it's turn comes. Since this does not invlove overheads of threads, it is expected to run faster than the code which generates the threads in the fly.

4 Code

The Code is attached in the tar file: $CS15B001.tar.gz$. Use the command $g++ -std = c++11 -pthread\ matrix\_multiplication.cpp$ to run the program. Input specifications are mentioned in the program. The code is included below for sake of

```

#include "ctpl_stl.h"
#include <bits/stdc++.h>

using namespace std;

vector< vector<int> > a;
vector< vector<int> > b;
vector< vector<int> > c;
int n,m,p;

struct Indices {
    Indices(int i, int j) { this->i = i; this->j = j; /*std::cout << "
        Indices ctor " << this->v << '\n';*/ }
    ~Indices() { /*std::cout << "Indices dtor\n";*/ }
    int i;
    int j;
};

void matrix_mult(int id, Indices& ind)
{
   c[ind.i][ind.j] = 0;
   for(int k=0;k<m;++k)
   {
      c[ind.i][ind.j] += a[ind.i][k]*b[k][ind.j];
   }
}
```

```

int main()
{
    //  Initialize  the thread pools
    int num_of_threads = 3;
    ctpl :: thread_pool pp(num_of_threads);

    // Take matrice indices as input
    cin>>n>>m>>p;

    //  Initialize  the vectors with required space
    a. resize (n);
    for(int  i=0;i<n;++i)
    {
        a[ i ]. resize (m);
    }

    b. resize (m);
    for(int  i=0;i<p;++i)
    {
        b[ i ]. resize (p);
    }

    c. resize (n);
    for(int  i=0;i<p;++i)
    {
        c[ i ]. resize (p);
    }
    // Matrix  initialization  complete

    // Take input from stdin
    for(int  i=0;i<n;++i)
    {
        for(int  j=0;j<m;++j)
        {
            cin>>a[i][j ];
        }
    }

    for(int  i=0;i<m;++i)
    {
        for(int  j=0;j<p;++j)
        {
            cin>>b[i][j ];
        }
    }
    // Done taking inputs


    // Multiplty the matrices
    for(int i=0;i<n;++i)
    {
        for(int j=0;j<p;++j)
        {
            pp.push(matrix_mult, Indices(i,j));
            // matrix_mult(1, Indices(i,j));
            // cout<<"Value Here is: "<<c[i][j]<<"\n";
        }
    }

    // Display the result
    for(int i=0;i<n;++i)
    {
        for(int j=0;j<p;++j)
        {
            cout<<c[i][j]<<" ";
        }
        cout<<endl;
    }


}
```