CPSCI 351: Assignment #3

Monday, November 14, 2016

Huy 7:00pm

Christopher Grant, Jimi Vandermost
3726, 6915

Contents

Problem Statement

- The problem that this program solves is that of multiplying large dimensioned matrices of randomly assigned doubles ($0 < x < 1$ exclusive).
- The problem also includes trace calculation of the resultant matrix.

Design Description

- Asks the user for dimension and number of threads to use.
- Executes with multiple threads, given by the number given by the user.
- Computes the trace of the resultant (product) matrix on a single thread
- Prints the two randomly assigned matrices, resultant matrix, and the trace of the matrix to the user.

Detailed explanation of how the workers and product function work:

Hi Jimi

Linux and C Libary Function Listing

- stdio.h
- stdlib.h
- pthreads.h

Code

Listing 1: Matrix Multiplication of nxn matrices using p_threads

```
/*

    This program generates two random integer arrays of user given
    dimensions, and multiplies them using parallel processing, displaying
    the result.

    Programmers: Christopher Grant and
    Date: November 15, 2016
*/

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

// prototypes
void *prod_worker(void *arg);
void product(int id, int *rows, int dim, double **matrix_a,
                double **matrix_b, double **matrix_r);
void print_matrix(double **matrix, int dim);

/*
    This struct will be passed along with each thread

    id = the thread #
    dim = nxn dimension of the matrix
    m1 = matrix A
    m2 = matrix B
    mr = resultant matrix
    rows = used to determine which thread handles which row
*/
typedef struct {
    double **m1;
    double **m2;
    double **mr;
    int id;
    int dim;
    int *rows;
} s_param;


int main()
{
    int dim_var, threads_var, rem_thread_var, rem_dim_var;
    int thread_array[threads_var];
    double **matrix_a, **matrix_b, **matrix_r;
    pthread_t *thread;
```

```
    pthread_attr_t attr_var;
    long ret_val;
    void *status;
    // ask user for matrix dimension and num threads
    printf("Hello, what size matrices would you like? (nxn) ");
    scanf("%d", &dim_var);
    printf("How many threads should this program use? ");
    scanf("%d", &threads_var);

    srand48(time(NULL)); // seed rand with time

    // alloc the first dim
    matrix_a = malloc(dim_var  * sizeof(double *));
    matrix_b = malloc(dim_var  * sizeof(double *));
    matrix_r = malloc(dim_var  * sizeof(double *));

    // alloc the second dim
    for(int row = 0; row < dim_var; row++)
    {
        matrix_a[row] = malloc(dim_var  * sizeof(double *));
        matrix_b[row] = malloc(dim_var  * sizeof(double *));
        matrix_r[row] = malloc(dim_var  * sizeof(double *));
    }

    // fill matrices with rand doubles
    for(int row = 0; row < dim_var; row++)
    {
        for(int col = 0; col < dim_var; col++)
        {
            matrix_a[row][col] = drand48();
            matrix_b[row][col] = drand48();
            matrix_r[row][col] = 0;
        }
    }

    // thread allocation and attribute initialization
    thread = (pthread_t *) malloc(threads_var * sizeof(pthread_t));
    pthread_attr_init(&attr_var);
    pthread_attr_setdetachstate(&attr_var, PTHREAD_CREATE_JOINABLE);


    // use the two rem(aining) variables to determine how many rows each thread
    // will be handling. store the values in thread_array
    rem_thread_var = threads_var;
    rem_dim_var = dim_var;

    for(int i = 0; i < threads_var; i++)
    {
        thread_array[i] = rem_dim_var / rem_thread_var;
```

```
        rem_dim_var -= thread_array[i];
        --rem_thread_var;
    }

    // init our structure
    s_param *arg;
    arg = (s_param *) malloc(threads_var * sizeof(s_param));

    // loop through the number of threads setting each thread
    // with this to correctly calculate its slice of the matrix
    // this goes around the problem of pthreads, where you need
    // static variables. we can use a struct as our static variable
    for(int i = 0; i < threads_var; i++)
    {
        arg[i].id = i;
        arg[i].rows = thread_array;
        arg[i].dim = dim_var;
        arg[i].m1 = &matrix_a;
        arg[i].m2 = &matrix_b;
        arg[i].mr = &matrix_r;

        // spawn new thread
        if(ret_val = pthread_create(&thread[i], &attr_var, prod_worker, (void *)(arg +
        {
            fprintf(stderr, "error value %ld", ret_val);
            exit(1);
        }
        printf("Thread %d started. \n", i+1);
    }

    // loop through each thread, joining it back to main
    for(int i = 0; i < threads_var; i++)
    {
        ret_val = pthread_join(thread[i],&status);
        if(ret_val)
        {
            fprintf(stderr, "error value %ld", ret_val);
            exit(1);
        }
        printf("Thread %d completed. \n", i+1);
    }

    // print results
    printf("Matrix A: \n");
    print_matrix(matrix_a, dim_var);
    printf("Matrix B: \n");
    print_matrix(matrix_b, dim_var);
    printf("Resultant Matrix AB: \n");
    print_matrix(matrix_r, dim_var);
```

```
    // trace
    return 0;
}



void *prod_worker(void *arg)
{
    s_param *s = (s_param *)arg;
    product(s->id, (s->rows), s->dim, *(s->m1), *(s->m2), *(s->mr));
    pthread_exit(0);
}


void product(int id, int *rows, int dim, double **matrix_a,
                double **matrix_b, double **matrix_r)
{
    int start_row, finish_row;
    double sum;

    for(int i = 0; i < id; i++)
    {
        start_row += rows[i];
    }

    finish_row = start_row + rows[id];

    for(int row = start_row; row < finish_row; row++)
    {
        for(int col = 0; col < dim; col++)
        {
            for(int k = 0; k < dim; k++)
            {
                sum = sum + (matrix_a[row][k] * matrix_b[k][col]);
            }
            matrix_r[row][col] = sum;
        }
    }
}

void print_matrix(double **matrix, int dim)
{
    for(int row = 0; row < dim; ++row)
    {
        printf("| ");
        for(int col = 0; col < dim; ++col)
        {
```

```
            printf(" %f ", matrix[row][col]);
        }
        printf(" |\n");
    }
   printf("\n");
}
```

Screenshot

Conclusion

What we learned from this assignment is using rand48, pthreads, process/worker architecture, and how to calculate the trace of a matrix.