Text

Description automatically generatedA picture containing text

Description automatically generatedText

Description automatically generated with low confidence

Code Description:

Space for three 12x12 arrays (mat1, mat2, mat3) is allocated at compile time in the global namespace. The two input matrices are filled with data by fileToMatrix(), which takes the filenames specified in the command line arguments. Then there is a loop which creates 12 threads. Each thread is responsible for calculating one row of mat3.

Also of note is the method for implementing 2d matrices. There are several ways of doing this in c but I chose to use pointers to 1d arrays where the first 12 elements represent the first row of the matrix, the next 12 elements represent the second row of the matrix, and so on. I have a helper function getIndex() which converts 2d indices to 1d indices.

Efficiency:

As far as efficiency and cost goes, the efficiency of a multithreaded program improves

on the efficiency of a single threaded program only if multiple CPUs or cores are utilized.

So a program with 12 threads will be faster than a program with 1 thread assuming that the system has multiple cores and assuming that multiple cores are actually used by the program (it's not guaranteed). A program with 144 threads will be faster than a program with 12 threads only if there are more than 12 processing units and if more than 12 processing units are utilized by the program (I'm pretty sure most personal computers don't have more than 12 cores or CPUs.) Threads are lightweight so the cost of creating a thread is not high, but still I probably would not write a program with 144 threads for a personal computer because there is just no benefit to having that many threads.

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#define NUM\_THREADS 12

/\*

Create some empty 12x12 matrices and an array of thread identifiers

\*/

int mat1[12][12];

int mat2[12][12];

int mat3[12][12];

pthread\_t threads[12];

/\*

Create a list of structures to store parameters for threaded function

\*/

struct thread\_params {

int \*mat1;

int \*mat2;

int \*mat3;

int row;

} t\_params[12];

/\*

Function Declarations

\*/

void updateRowCol(int\* row, int \*col);

int\* fileToMatrix(char\* filename, FILE\* fp, int\* matrix);

int getIndex(int row, int col);

void rowTimesColumn(int \*mat1, int \*mat2, int\* mat3, int mat1row, int mat2col);

void\* calcOutputRow(void \*params);

void printMatrix(int\* matrix);

void printMatrix(int\* matrix);

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

int\* mat1ptr;

int\* mat2ptr;

FILE \*fp1, \*fp2;

if(argc!=3){

fprintf(stderr,"Usage: ./filename <mat1> <mat2>\n");

return -1;

}

mat1ptr = fileToMatrix(argv[1], fp1, (int\*)mat1);

mat2ptr = fileToMatrix(argv[2], fp2, (int\*)mat2);

int \*mat3ptr = (int\*)mat3;

// Creates 12 threads, each of which calculates one row of the output matrix

for (int i=0; i<12; i++) {

struct thread\_params t\_params = {mat1ptr, mat2ptr, mat3ptr, i};

pthread\_create(&threads[i], NULL, calcOutputRow, (void \*)&t\_params);

pthread\_join(threads[i], NULL);

}

printf("Printing mat1\n\n");

printMatrix(mat1ptr);

printf("Printing mat2\n\n");

printMatrix(mat2ptr);

printf("Printing mat3\n\n");

printMatrix(mat3ptr);

printf("Exiting from main\n\n");

pthread\_exit(NULL);

return 0;

}

// Takes a filename (for a text file), an empty file pointer, and an empty matrix

// and returns a filled matrix.

// Assumes that matrix entries are delimited by spaces (any number) and that

// matrix rows are delimited by newlines.

int\* fileToMatrix(char\* filename, FILE\* fp, int\* matrix) {

int c, number, index;

int row = 0, col = 0;

fp = fopen(filename, "r");

while ((c = fgetc(fp)) != EOF) {

if (c != ' ') {

number = c - '0';

index = getIndex(row, col);

matrix[index] = number;

updateRowCol(&row, &col);

}

}

return matrix;

}

// Takes a matrix index and returns the next index.

// For example 0,1 becomes 0,2 and 1,11 becomes 2,0

void updateRowCol(int\* row, int \*col) {

if (\*col >= 12) {

\*col = 0;

\*row = \*row + 1;

} else {

\*col = \*col+1;

}

}

// This function is very important. It takes the row and column numbers

// of a matrix (0 indexed) and returns the corresponding flat array index.

// I use this method for matrix indexing because in my implementation

// I choose to pass matrices to functions as type int\* (rather than int\*\*).

int getIndex(int row, int col) {

return row\*12+col;

}

// Takes two input matrices 'mat1', 'mat2', an output matrix 'mat3',

// a row 'mat1row' and a column 'mat2col'. It takes the dot product of

// the column and row indicated by 'mat1row' and 'mat2col'

// and stores it in the proper index of mat3

void rowTimesColumn(int \*mat1, int \*mat2, int\* mat3, int mat1row, int mat2col) {

int sum = 0, index1, index2, index3;

for (int i=0; i<12; i++) {

index1 = getIndex(mat1row, i);

index2 = getIndex(i, mat2col);

sum = sum + mat1[index1]\*mat2[index2];

}

index3 = getIndex(mat1row, mat2col);

mat3[index3] = sum;

}

// A threaded function to calculate one row of the output of the dot product

// of two matrices. The row number is specified in 'params'. So are the

// two input matrices and the output matrix. Essentially we iterate through

// the columns of 'mat2' and take the dot product with the indicated row of 'mat1'.

void\* calcOutputRow(void \*params) {

struct thread\_params \*p = (struct thread\_params\*)params;

printf("Creating thread: %d\n", p->row);

for (int col=0; col<12; col++) {

rowTimesColumn(p->mat1, p->mat2, p->mat3, p->row, col);

}

printf("Exiting thread: %d\n\n", p->row);

pthread\_exit(NULL);

}

// Pretty prints a matrix to the terminal

void printMatrix(int\* matrix) {

int index;

for (int row=0; row<12; row++) {

for (int col=0; col<12; col++) {

index = getIndex(row, col);

printf("%d ", matrix[index]);

}

printf("\n");

}

printf("\n");

}