Samuel Lichtenheld
3/17/17

Comments

Parts 1 and 3 were performed on a desktop computer to eliminate the overhead of using the HiPerGator. An Intel i7 6700k processor with 4 cores and 8 threads with hyperthreading was used. Thus, 1,2,4 and 8 threads were tested for these parts. Every option was run 5 times and the average was taken therefrom. All code will be put at the bottom to make the assignment easier to read.

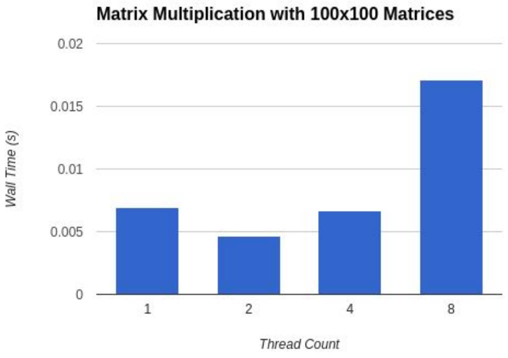
The values for Part 2 were also taken from using my desktop. I was receiving the following error when using the hipergator:

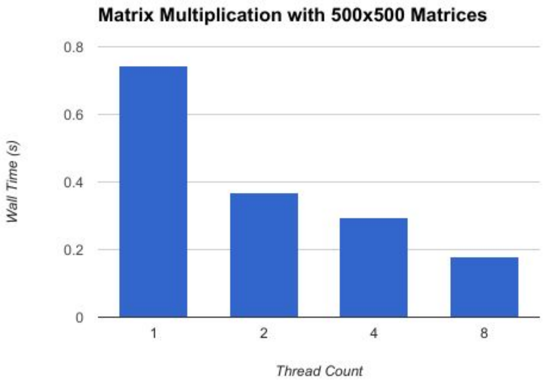
error while loading shared libraries: libiomp5.so: cannot open shared object file: No such file or directory

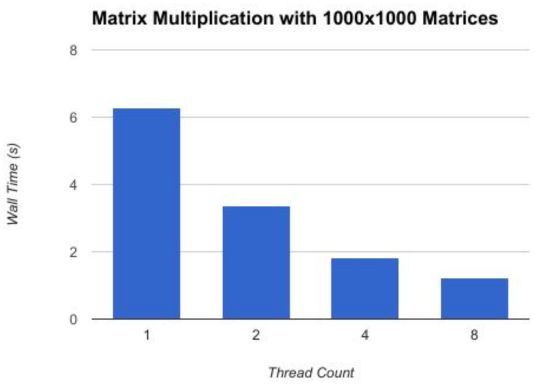
I was able to run Part 2 using for tasks with a variable number of threads.

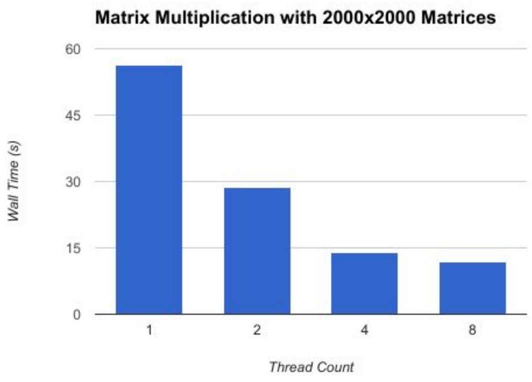
Part 1

Here are four bar graphs representing the time of execution with matrix sizes of 100x100, 500x500, 1000x1000, and 2000x2000 respectively.









With the 100x100 matrix being the exception, every other test experienced significant speedup with increased thread count. Intel's hyper-threading (8 threads) also demonstrated much speedup with the exception of the 100x100 matrix, which only benefited from 2 threads. For the most part, speedup was directly proportional to the number of threads being used.

Part 2

I decomposed part2 as follows:

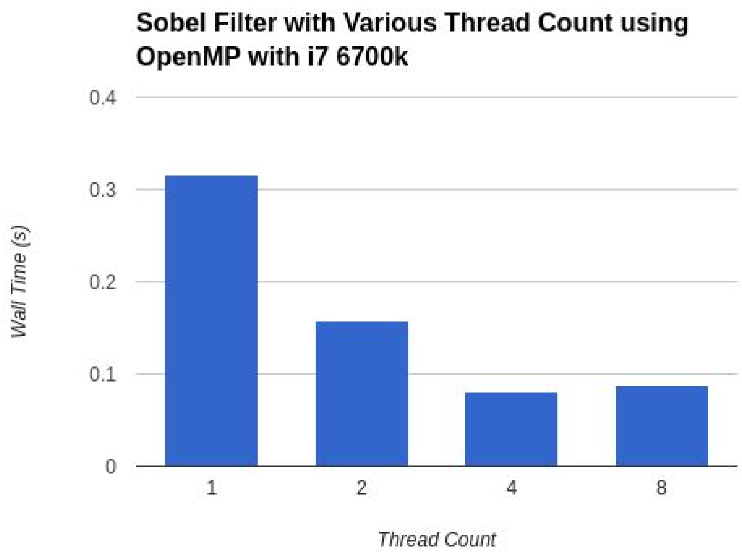
- Every node would get a copy of matrix B in the calculation A*B (through MPI_Bcast)
- Every node would get $A_{number\_of\_rows}$/total_ranks rows from matrix A
    - the root node would also get the leftover $A_{number\_of\_rows}$%total_ranks from matrix A
- The root node would then collect all the results from every node

Below is a chart comparing matrix multiplication using only OpenMP, only OpenMPI, and a combination of the two. While the hybrid approach generated lackluster results, this is due to the fact that each task was already running on an individual CPU core. Thus, using the HiperGator is the only way to realize these gains. However, I did not leave myself enough time to test it on the HiperGator and all the nodes were being used by other students running large tasks.

	4 Threads (OPENMP)	4 Tasks (OPENMP)I	4 Tasks, 4 Threads (HYBRID)
100x100	0.0066416	0.003361	0.048394
500x500	0.2951254	0.221778	0.231552
1000x1000	1.8086086	1.816503	1.59321
2000x2000	14.071474	17.43091	16.03788

Part 3

As the sobel filter execution time does not depend on the image being used, a randomly populated 3840x2160 8-bit array was used.



Here, proportional speedup was achieved with every additional real core being used. However, when hyperthreading is used (8 threads), there is actually a slowdown. This could be due to all the integer adder functional units being fully saturated with 4 threads.

Appendix (Code)

Part 1

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

```
#include <stdlib.h>
#include "randNumGen.h"
#include "execTime.h"

void mtrx_populate(double* mtrx, int rows, int cols);
void mtrx_print(double* mtrx, int rows, int cols);
void mtrx_multiply(double* A, double* B, double* AB, int A_rows, int A_cols, int B_cols);

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

        if (argc != 5) {
                perror("ERROR: ./program < A_rows > < A_cols > < B_cols > < numThreads>
");
                exit(EXIT_FAILURE);
        }
        //printf("num of threads: %d\n", omp_get_num_threads());

        int A_rows = strtol(argv[1],NULL,10);
        int A_cols = strtol(argv[2],NULL,10);
        int B_rows = A_cols;
        int B_cols = strtol(argv[3],NULL,10);
        int numThreads = strtol(argv[4],NULL,10);

        double* A_mtrx = malloc(A_rows*A_cols*sizeof(double));
        double* B_mtrx = malloc(B_rows*B_cols*sizeof(double));
        double* r_mtrx = malloc(A_rows*B_cols*sizeof(double));

        /* populate matrices */
        mtrx_populate(A_mtrx,A_rows,A_cols);
        mtrx_populate(B_mtrx,B_rows,B_cols);

        //mtrx_print(A_mtrx,A_rows,A_cols);
        //mtrx_print(B_mtrx,B_rows,B_cols);

        omp_set_num_threads(numThreads);

        startTimer();
        mtrx_multiply(A_mtrx, B_mtrx, r_mtrx, A_rows, A_cols, B_cols);
        stopTimer();

        printf("EXECTIME: %f\n", getTimeSpent());

        //mtrx_print(r_mtrx,A_rows,B_cols);
```

```
    return 0;

}

void mtrx_populate(double* mtrx, int rows, int cols){ // populates whole row, then moves to next row
    for(int i = 0; i < rows ; i++) {
        for (int j = 0; j < cols; j++ ){
            mtrx[i*cols + j] = randinrangeDOUBLE(0,10000);
            //printf("%d ", mtrx[i*cols + j] );
        }
    }
    //printf("\n");
}

void mtrx_print(double* mtrx, int rows, int cols){
    for(int i = 0; i < rows ; i++) {
        for (int j=0; j < cols ; j++) {
            printf("%f ", mtrx[i*cols + j]);
        }
        printf("\n");
    }
    printf("\n");
}

void mtrx_multiply(double* A, double* B, double* AB, int A_rows, int A_cols, int B_cols){
    int i, j, k;

    #pragma omp parallel shared(A,B,AB) private(i,j,k)
    //printf("num of threads: %d\n", omp_get_num_threads());
    {
    #pragma omp for schedule(static)
        for (i = 0; i < A_rows ; i++ ) {
            for (j = 0; j < B_cols; j++ ) {
                AB[i*B_cols + j] = 0;
                for (k = 0; k < A_cols ; k++ ){
                    AB[i*B_cols + j] += A[i*A_cols+k]*B[k*B_cols+j];
                }
            }
        }
    }
}
```

PART2

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

// SHOULD BELONG IN SEPARATE C FILES
double begin;
double end;
void startTimer(){
	begin = MPI_Wtime();
}
void stopTimer(){
	MPI_Barrier(MPI_COMM_WORLD); /* IMPORTANT */
	end = MPI_Wtime();
}
double getTimeSpent(){
	return (double)(end - begin);
}
double randinrangeDOUBLE(float HI, float LO) {
	return LO + (double)(rand()) / ( (double)(RAND_MAX/(HI-LO)));
}

//*******************************************************

void mtrx_populate(double* mtrx, int rows, int cols);
void mtrx_populate_simple(double* mtrx, int rows, int cols);
void mtrx_print(double* mtrx, int rows, int cols);
void mtrx_multiply(double* A, double* B, double* AB, int A_rows, int A_cols, int B_cols);
int xy2int(int x, int y, int cols);


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

	MPI_Status status;
	int my_rank, total_ranks;
	int root = 0;
	int tag = 0;

	if (argc != 5) {
		perror("ERROR: ./program < A_rows > < A_cols > < B_cols > < numThreads>
");
        exit(EXIT_FAILURE);
    }

    // MPI BEGIN
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&total_ranks);

    // Every node needs access to rows and cols of each input matrix
    int A_rows = strtol(argv[1],NULL,10);
    A_rows+= (total_ranks - (A_rows%total_ranks)); // BOOTLEGGED FIX FOR ISSUE
    int A_cols = strtol(argv[2],NULL,10);
    int B_rows = A_cols;
    int B_cols = strtol(argv[3],NULL,10);
    int numThreads = strtol(argv[4],NULL,10);

    omp_set_num_threads(numThreads);

    //omp_set_num_threads(numThreads);
    startTimer();

    int num_rows_per_proc = A_rows/total_ranks;
    int stride = num_rows_per_proc*A_cols; // A_cols = elements per row
    int left_over_rows = A_rows%total_ranks;
    int left_over_elems = left_over_rows*A_cols;

    int A_recv_bufSIZE = stride+A_cols;
    int R_bufSIZE = (num_rows_per_proc)*B_cols;
    if (left_over_rows) R_bufSIZE+=B_cols;

    double* A_recv_buf = (double *)malloc( (A_recv_bufSIZE)*(sizeof(double)) );
    double* R_buf = (double *)malloc( R_bufSIZE*(sizeof(double)));


    int* sendcounts = (int *)malloc(total_ranks*sizeof(int));
    int* recvcounts = (int *)malloc(total_ranks*sizeof(int));
    int* displs = (int *)malloc(total_ranks*sizeof(int));
    int* recv_displs = (int *)malloc(total_ranks*sizeof(int));

    int sum = 0;
    int recv_sum = 0;
    for (int i = 0; i < total_ranks; i++) {
        sendcounts[i] = stride;
        if (left_over_rows > 0) {
            sendcounts[i] += A_cols;
            left_over_rows--;
        }
        displs[i] = sum;
        sum+=sendcounts[i];

        recvcounts[i] = sendcounts[i]/A_cols*B_cols;
        recv_displs[i] = recv_sum;
        recv_sum+=recvcounts[i];
    }

    // if (my_rank == root){
    //      printf("TOTAL RANKS: %d\n", total_ranks );
    //      printf("A_recv_bufSIZE: %d R_bufSIZE: %d \n", R_bufSIZE, A_recv_bufSIZE);
    //      printf("R_MATRIX_SIZE: %d \n", A_rows*B_cols);
    //      for (int i = 0; i < total_ranks; i++) {
    //          printf("%d: sendcount: %d displs: %d recvcount: %d recv_displs: %d
\n", i, sendcounts[i], displs[i], recvcounts[i], recv_displs[i]);
    //      }
    // }

    double* B_mtrx = (double *)malloc(B_rows*B_cols*sizeof(double));

    // A_mtrx and R_mtrx only used by root
    double* A_mtrx;  // row major order
    double* R_mtrx;

    // Populate A and B, instantiate R_mtrx in root
    if(my_rank==root) {

        mtrx_populate_simple(B_mtrx,B_rows,B_cols); // TODO: change back to
normal
        //mtrx_print(B_mtrx, B_rows, B_cols);

        A_mtrx = (double *)malloc(A_rows*A_cols*sizeof(double));
        mtrx_populate_simple(A_mtrx,A_rows,A_cols); // TODO: change back to
normal

        //mtrx_print(A_mtrx, A_rows, A_cols);
        R_mtrx = (double *)malloc(A_rows*B_cols*sizeof(double));
    }

    // broadcast B_mtrx to all processes
```

```
        MPI_Bcast(B_mtrx, B_rows*B_cols, MPI_DOUBLE,root , MPI_COMM_WORLD);

        // scatter A to all processes
        MPI_Scatterv(A_mtrx, sendcounts, displs,
            MPI_DOUBLE, A_recv_buf, A_recv_bufSIZE,
            MPI_DOUBLE,
            root, MPI_COMM_WORLD);

        // for(int i = 0; i < sendcounts[my_rank]; i++) {
        //          printf("%d ", (int)A_recv_buf[i]);
        // }
        // printf("\n");

        // matrix multiplication
        mtrx_multiply(A_recv_buf, B_mtrx, R_buf, sendcounts[my_rank]/A_cols, A_cols,
B_cols );

        MPI_Gatherv(R_buf, R_bufSIZE, MPI_DOUBLE,
           R_mtrx, recvcounts, recv_displs,
           MPI_DOUBLE, root, MPI_COMM_WORLD);

        stopTimer();
        if (my_rank == root) {
                printf("WALL TIME: %f\n",getTimeSpent() );
                // mtrx_print(R_mtrx, A_rows, B_cols);
        }


        MPI_Finalize();

        return 0;

}

void mtrx_multiply(double* A, double* B, double* AB, int A_rows, int A_cols, int B_cols){
        int i, j, k;

        #pragma omp parallel shared(A,B,AB) private(i,j,k)
        //printf("num of threads: %d\n", omp_get_num_threads());
        {
        #pragma omp for schedule(static)
                for (i = 0; i < A_rows ; i++ ) {
                        for (j = 0; j < B_cols; j++ ) {
                                AB[i*B_cols + j] = 0;
                                for (k = 0; k < A_cols ; k++ ){
                                        AB[i*B_cols + j] += A[i*A_cols+k]*B[k*B_cols+j];
                                }
                        }
                }
        }
}

void mtrx_populate(double* mtrx, int rows, int cols){ // populates whole row, then moves to next row
        for(int i = 0; i < rows ; i++) {
                for (int j = 0; j < cols; j++ ){
                        mtrx[i*cols + j] = randinrangeDOUBLE(0,10000);
                }
        }
}

void mtrx_populate_simple(double* mtrx, int rows, int cols){
        for(int i = 0; i < rows ; i++) {
                for (int j = 0; j < cols; j++ ){
                        mtrx[i*cols + j] = i*cols + j;
                }
        }
}

void mtrx_print(double* mtrx, int rows, int cols){
        for(int i = 0; i < rows ; i++) {
                for (int j=0; j < cols ; j++) {
                        printf("%d ", (int)mtrx[i*cols + j]);
                }
                printf("\n");
        }
        printf("\n");
}

int xy2int(int x, int y, int cols){
        return y*cols+x;
}
```

PART3

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

```
#include <stdint.h>
#include <omp.h>
#include <math.h>

#include <stdlib.h>
#include "randNumGen.h"
#include "execTime.h"

#define IMAGEWIDTH 3840 //3840
#define IMAGEHEIGHT 2160//2160

void mtrx_populate(int8_t* mtrx, int rows, int cols);
void mtrx_print(int8_t* mtrx, int rows, int cols);
void sobel_conv(int8_t* in_mtrx, int8_t* out_mtrx, int rows, int cols);

int xy2int(int x, int y, int cols);

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

    if(argc != 2) {
        perror("ERROR: ./program < numThreads > ");
        exit(EXIT_FAILURE);
    }
    // take in number of threads
    int numThreads = strtol(argv[1],NULL,10);


    // make 2k image 3840 x 2160 with 8 bits each, one for input, one for output
    int8_t* in_mtrx = malloc(IMAGEWIDTH*IMAGEHEIGHT*sizeof(int8_t));
    int8_t* sbl_mtrx = malloc(IMAGEWIDTH*IMAGEHEIGHT*sizeof(int8_t));

    // randomize input (fine for measuring performance since not real program)
    mtrx_populate(in_mtrx, IMAGEHEIGHT, IMAGEWIDTH);
    //mtrx_print(in_mtrx, IMAGEHEIGHT, IMAGEWIDTH);
    omp_set_num_threads(numThreads);

    startTimer();
    sobel_conv(in_mtrx, sbl_mtrx, IMAGEHEIGHT, IMAGEWIDTH);
    stopTimer();
    //mtrx_print(sbl_mtrx, IMAGEHEIGHT, IMAGEWIDTH);

    printf("EXECTIME: %f\n", getTimeSpent());
    // sobel filter
}
```

```
void sobel_conv(int8_t* in_mtrx, int8_t* out_mtrx, int rows, int cols){

        // zero out output matrix edges
        for (int i = 0; i < cols; i++ ) {
                out_mtrx[xy2int(i,0,cols)] = 0;
                out_mtrx[xy2int(i,rows-1,cols)] = 0;
        }
        for (int i = 1; i < rows-1; i++) {
                out_mtrx[xy2int(0,i,cols)] = 0;
                out_mtrx[xy2int(cols-1,i,cols)] = 0;
        }

        int i, j;
        #pragma omp parallel shared(in_mtrx,out_mtrx) private(i,j)
        {
                #pragma omp for schedule(static)
                for (j = 1; j < rows-1; j++ ) {
                        for (i = 1; i < cols-1; i++) {

                                int gX = 0;
                                gX += in_mtrx[                xy2int( i-1,     j-1      ,
cols)];
                                gX += 2*in_mtrx[        xy2int( i-1,     j        ,        cols)];
                                gX += in_mtrx[          xy2int( i-1,     j+1      ,        cols)];

                                gX -= in_mtrx[          xy2int( i+1,     j-1      ,        cols)];
                                gX -= 2*in_mtrx[        xy2int( i+1,     j        ,        cols)];
                                gX -= in_mtrx[          xy2int( i+1,     j+1      ,        cols)];

                                int gY = 0;
                                gY = in_mtrx[           xy2int( i-1,     j-1      ,        cols)];
                                gY += 2*in_mtrx[        xy2int( i,               j-1      ,
cols)];
                                gY += in_mtrx[          xy2int( i+1,     j-1      ,        cols)];

                                gY -= in_mtrx[          xy2int( i-1,     j+1      ,        cols)];
                                gY -= 2*in_mtrx[        xy2int( i,               j+1      ,
cols)];
                                gY -= in_mtrx[          xy2int( i+1,     j+1      ,        cols)];

                                //printf("%d,%d\n",gX,gY );

                                int temp = sqrt((double)(gX*gX+gY*gY));
                if (temp > 127) out_mtrx[xy2int(i,j,cols)] = 127;
                else if (temp < -128) out_mtrx[xy2int(i,j,cols)] = -128;
                else out_mtrx[xy2int(i,j,cols)] = temp;

                //out_mtrx[xy2int(i,j,cols)] = (int8_t)sqrt((double)(gX*gX+gY*gY));
            }
        }
    }
}

void mtrx_populate(int8_t* mtrx, int rows, int cols){ // populates whole row, then moves to next row
    for(int i = 0; i < rows ; i++) {
        for (int j = 0; j < cols; j++ ){
            mtrx[i*cols + j] = randinrangeINT8(-128,127);
        }
    }
}

void mtrx_print(int8_t* mtrx, int rows, int cols){
    for(int i = 0; i < rows ; i++) {
        for (int j=0; j < cols ; j++) {
            printf("%d ", mtrx[i*cols + j]);
        }
        printf("\n");
    }
    printf("\n");
}

int xy2int(int x, int y, int cols){
    return y*cols+x;
}
```