**Project Report**

Todd Dick

CPE 412 Fall 16

Instructor: Dr. Wells

**Problem Introduction**

Digital media files often need to be altered or have in-depth analysis done on them (e.g., stegography, signal analysis, FFT’s, filtering, etc.). Sometimes, batch processing is the most efficient way to do it. Parallelization of data input and output is problematic, however. We can’t split input onto multiple threads using pthreads, for example. We could not guarantee the integrity of the input. If we did, we’d make the input read serial for all intents and purposes with mutual exclusion. We end up with one thread or process to read in a file. But can parallelization be applied successfully to read in multiple files, then perform some algorithm on the data from those files? If so, which programming paradigm is best for it?

I wrote 4 programs for this project. I wrote a serial program that reads in 10 media files into vector data structures that could be used to perform any type of algorithm one might want to perform on them. C/C++ provides no way to dynamically name a vector, so I used global vectors named file0-file9. The program times the execution of reading the files, and stops after reading the data into vectors. I then wrote a version of the program in OpenMPI, OpenMP, and using pthreads.

I ran all 4 programs on the Jetson cluster for performance data. In my original proposal, I was going to use my Mac, but I couldn’t get the OpenMP or OpenMPI libraries to link on my Mac, so all data comes from the Jetson cluster. My original proposal was to decide how many .png files it would take to make a cluster computer worth using over my Mac. It will be made clear in this report why that is irrelevant. The real question has already been stated. Which programming paradigm best serves the base algorithm?

**Parallelization Approach Employed**

I used OpenMPI, OpenMP and pthreads to parallelize a batch read. The intent was to get speedup.

**Empirical Method and Results**

I used a .mp3 file from a CPE495 lecture to get significant execution times. The .png file I had originally intended to use did not give significant execution time for the reading algorithm.

For the OpenMPI implementation, I did not use a gather function. I was only scattering work, so I got execution times for each process. A gather would have given me no functionality. For the pthread and OpenMP implementations, I used a standard output to make sure the number of threads I was using spawned, but I am not sure I got execution times for individual threads.

The following tables show all execution times for each paradigm. Times were rounded to nearest 1000th for formatting purposes.

**Tables**

For serial execution, I wrote a script that ran the serial program five times on 10 files. I ran the same script on 1 file. The script was executed on 1 Jetson node.

|  |  |  |
| --- | --- | --- |
| **# of files** | **1** | **10** |
| **Execution time in seconds** | 1.851 | 18.531 |

Table . Serial implementation execution times. Time shown is fastest execution

For MPI, I wrote a script that executed the program three times. The script was executed on 1-10 Jetson nodes.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **# of processes** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **Execution time in s** | 17.925 | 9.008 | 7.228 | 5.378 | 3.583 | 3.586 | 3.646 | 3.519 | 3.64 | 1.757 |

Table . MPI implementation execution times. Time shown is fastest process.

For OpenMP, I wrote a script that executed the program with 1-10 threads. I ran the script twice on 1 Jetson node.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **# of threads** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **Execution time in s** | 18.793 | 9.819 | 8.608 | 7.868 | 7.042 | 7.247 | 7.981 | 7.891 | 7.983 | 7.178 |

Table . OMP implementation execution times. Time shown is fastest recorded.

For pthreads, I modified the same script from the OMP test. The script was run twice on 1 Jetson node.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **# of processes** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **Execution time in s** | 17.549 | 9.221 | 8.001 | 7.522 | 7.095 | 7.139 | 7.445 | 7.999 | 7.943 | 7.194 |

Table . pthreads implementation execution times. Time shown is fastest recorded.

MPI Speedup

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **# of processes** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **Speedup in s** | 1.034 | 2.057 | 2.564 | 3.446 | 5.172 | 5.168 | 5.083 | 5.266 | 5.091 | 10.547 |

Table . Speedup for MPI implementation.

OpenMP Speedup

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **# of threads** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **Execution time in s** | 0.986 | 1.887 | 2.153 | 2.355 | 2.631 | 2.557 | 2.323 | 2.348 | 2.321 | 2.582 |

Table . Speedup for OMP implementation.

Pthreads Speedup

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **# of processes** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **Execution time in s** | 1.056 | 2.01 | 2.316 | 2.464 | 2.612 | 2.596 | 2.489 | 2.317 | 2.323 | 2.576 |

Table . Speedup for pthreads implementation.

**General Conclusions**

The results were surprising. OpenMPI was the only paradigm that worked as one might expect. The MPI program spreads the files for reading out to however many processes we use up to 10. With 10 processes, the execution time per process is roughly the same as it was for one file applied to the serial program (see Tables 1 and 2). With OpenMP and pthreads, I got the highest speedup when each thread read in two files, and no significant speedup vs. MPI with 3 processes, regardless of the number of threads deployed.

Oddly, MPI\_Finalize() did not shut down all of the processes, and the execution time was reported for the number of processes used, with faster times –not shown- in some results showing execution times of the individual processes in the load balanced scatterv (see Appendix – read\_mpi.cpp). This could indicate that the results for the MPI implementation are skewed due to the lack of overhead from a gather operation, but even if that were implemented, one would only expect a small addition to the execution times. In Table 5, we can see that there was no speedup where load balancing was an issue. The execution times reflected processes 5-9 running the read function twice on at least one process.

I don’t know the explanation for the lack of speedup in the OMP and pthreads implementations. Overhead from join operations should not negate all speedup from deploying multiple threads.

In conclusion, the size of the files affects execution time as a direct function of size. The larger the files, the larger the execution time. The number of files being processed affects execution time in the same way. The paradigm used for parallelization is the main factor in speedup vs. serial. OpenMPI is the superior paradigm, acting as one would expect. For batch processing of media files, MPI should be used, and due to the lack of unnecessary overhead from thread spawning and joining operations, allows much more flexibility. MPI could even be used for real-time algorithms.

**Code Appendix**

//

//  read\_serial.cpp

//  412project

//

//  Created by Todd Dick on 12/16.

//  Copyright © 2016 Todd Dick. All rights reserved.

//

/\* This program will read in 10 .png files and write the data byte by byte to separate vector data type containers to be used for whatever purposes. This program is designed to be a baseline program to measure the time it takes to read in 10 media files. This program can be extended to do something with the vectors, but that is outside of the scope of this project.\*/

/\* There is no way to dynamically name vectors in C or C++, so this program will only use 10 files. To use more, more vectors need to be declared.\*/

/\* Prior to using this program, enter the following from command line in the directory containing your 10 media files. "ls \*.<filetype> > list.txt" without the quotes.\*/

/\* to compile --- g++ read\_serial.cpp -o serial -O3 -std=c++11 \*/

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstdint>

#include <vector>

#include <fstream>

#include <string>

#include <cstring>

#include <sys/time.h>

using namespace std;

#define TIMER\_CLEAR     (tv1.tv\_sec = tv1.tv\_usec = tv2.tv\_sec = tv2.tv\_usec = 0)

#define TIMER\_START     gettimeofday(&tv1, (struct timezone\*)0)

#define TIMER\_ELAPSED   ((long long) (tv3.tv\_usec)+((long long) (tv3.tv\_sec)\*1000000))

#define TIMER\_STOP      {gettimeofday(&tv2, (struct timezone\*)0);timersub(&tv2,&tv1,&tv3);}

struct timeval tv1,tv2,tv3;

vector<uint8\_t> file0;

vector<uint8\_t> file1;

vector<uint8\_t> file2;

vector<uint8\_t> file3;

vector<uint8\_t> file4;

vector<uint8\_t> file5;

vector<uint8\_t> file6;

vector<uint8\_t> file7;

vector<uint8\_t> file8;

vector<uint8\_t> file9;

void MakeList(vector<string> &list){

    ifstream infile;

    string line;

    infile.open("list.txt");

    if (!infile){

        cout << "error opening list.txt file" << endl;

        exit(1);

    }

    while(getline(infile,line)){

        list.push\_back(line);

    }

    infile.close();

}

// Hybrid of C/C++ methods for reading binary, because Jetson does not support C++14 and those are the methods I originally used.

void MakeVector(int number, vector<string> &list){

    FILE \* infile;

    string tempStr = list[number];

    const char\* filePath = tempStr.c\_str();

    infile = fopen(filePath,"rb");

    fseek(infile,0L,SEEK\_END);

    uint64\_t fileSize = ftell(infile);

    fseek(infile,0L,SEEK\_SET);

    uint8\_t buff[1] = {0};

    int i = 0;

    while( i < fileSize ){

        size\_t readStuff = fread(&buff,1,sizeof(uint8\_t),infile);

        switch(number){

            case 0:

                file0.push\_back(buff[0]);

                break;

            case 1:

                file1.push\_back(buff[0]);

                break;

            case 2:

                file2.push\_back(buff[0]);

                break;

            case 3:

                file3.push\_back(buff[0]);

                break;

            case 4:

                file4.push\_back(buff[0]);

                break;

            case 5:

                file5.push\_back(buff[0]);

                break;

            case 6:

                file6.push\_back(buff[0]);

                break;

            case 7:

                file7.push\_back(buff[0]);

                break;

            case 8:

                file8.push\_back(buff[0]);

                break;

            case 9:

                file9.push\_back(buff[0]);

                break;

            default:

                break;

        }

        i++;

    }

    fclose(infile);

}

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

    vector<string> list;

    int i;

    MakeList(list);

    TIMER\_CLEAR;

    TIMER\_START;

    for (i = 0; i < list.size(); i++){

        MakeVector(i,list);

    }

    TIMER\_STOP;

    cout << "Execution time = " << TIMER\_ELAPSED/1000000.0 << endl;

    //Do something with the vectors...

    return 0;

}

//

//  read\_mpi.cpp

//  412project

//

//  Created by Todd Dick on 12/16.

//  Copyright © 2016 Todd Dick. All rights reserved.

//

/\* This program will read in 10 .png files and write the data byte by byte to separate vector data type containers to be used for whatever purposes. This program is designed to be a baseline program to measure the time it takes to read in 10 media files. This program can be extended to do something with the vectors, but that is outside of the scope of this project.\*/

/\* There is no way to dynamically name vectors in C or C++, so this program will only use 10 files. To use more, more vectors need to be declared.\*/

/\* Prior to using this program, enter the following from command line in the directory containing your 10 media files. "ls \*.<filetype> > list.txt" without the quotes.\*/

/\* to compile --- mpic++ read\_mpi.cpp -o mpi -O3 -std=c++11 \*/

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstdint>

#include <vector>

#include <fstream>

#include <string>

#include <cstring>

#include <sys/time.h>

#include <mpi.h>

using namespace std;

#define TIMER\_CLEAR     (tv1.tv\_sec = tv1.tv\_usec = tv2.tv\_sec = tv2.tv\_usec = 0)

#define TIMER\_START     gettimeofday(&tv1, (struct timezone\*)0)

#define TIMER\_ELAPSED   ((long long) (tv3.tv\_usec)+((long long) (tv3.tv\_sec)\*1000000))

#define TIMER\_STOP      {gettimeofday(&tv2, (struct timezone\*)0);timersub(&tv2,&tv1,&tv3);}

struct timeval tv1,tv2,tv3;

vector<uint8\_t> file0;

vector<uint8\_t> file1;

vector<uint8\_t> file2;

vector<uint8\_t> file3;

vector<uint8\_t> file4;

vector<uint8\_t> file5;

vector<uint8\_t> file6;

vector<uint8\_t> file7;

vector<uint8\_t> file8;

vector<uint8\_t> file9;

// Function to create a vector of filenames

void MakeList(vector<string> &list){

    ifstream infile;

    string line;

    infile.open("list.txt");

    if (!infile){

        cout << "error opening list.txt file" << endl;

        exit(1);

    }

    while(getline(infile,line)){

        list.push\_back(line);

    }

    infile.close();

}

// Hybrid of C/C++ methods for reading binary, because Jetson does not support C++14 and those are the methods I originally used.

void MakeVector(int number, vector<string> &list){

    FILE \* infile;

    string tempStr = list[number];

    const char\* filePath = tempStr.c\_str();

    infile = fopen(filePath,"rb");

    fseek(infile,0L,SEEK\_END);

    uint64\_t fileSize = ftell(infile);

    fseek(infile,0L,SEEK\_SET);

    uint8\_t buff[1] = {0};

    int i = 0;

    while( i < fileSize ){

        size\_t readStuff = fread(&buff,1,sizeof(uint8\_t),infile);

        switch(number){

            case 0:

                file0.push\_back(buff[0]);

                break;

            case 1:

                file1.push\_back(buff[0]);

                break;

            case 2:

                file2.push\_back(buff[0]);

                break;

            case 3:

                file3.push\_back(buff[0]);

                break;

            case 4:

                file4.push\_back(buff[0]);

                break;

            case 5:

                file5.push\_back(buff[0]);

                break;

            case 6:

                file6.push\_back(buff[0]);

                break;

            case 7:

                file7.push\_back(buff[0]);

                break;

            case 8:

                file8.push\_back(buff[0]);

                break;

            case 9:

                file9.push\_back(buff[0]);

                break;

            default:

                break;

        }

        i++;

    }

    fclose(infile);

}

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

    //generically useful variable

    int i;

    //get a vector of the filenames

    vector<string> list;

    MakeList(list);

    //set up for MPI

    int numtasks, rank;

    MPI\_Status status;

    MPI\_Init(NULL,NULL); // initalize MPI environment

    MPI\_Comm\_size(MPI\_COMM\_WORLD,&numtasks); // get total number of MPI processes

    MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank); // get unique task id number

    if(numtasks > 10){

        cout << "Number of processes must be <= 10." << endl;

        exit(1);

    }

    int position[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    int sendcounts[10];

    int displs[10];

    int localSize = 10/numtasks;

    int remainder = 10 % numtasks;

    unsigned int sum = 0;

    for (i = 0; i<numtasks; i++){

        sendcounts[i] = localSize;

        if (remainder > 0){

            sendcounts[i] = sendcounts[i] + 1;

            remainder--;

        }

        displs[i] = sum;

        sum += sendcounts[i];

    }

    //start timer

    TIMER\_CLEAR;

    TIMER\_START;

    int buffer[sendcounts[rank]];

    MPI\_Scatterv(position,sendcounts,displs,MPI\_INT, buffer,sendcounts[rank],MPI\_INT,0,MPI\_COMM\_WORLD);

    for (i = 0; i < sendcounts[rank]; i++){

        int number = buffer[i];

        MakeVector(number,list);

    }

    TIMER\_STOP;

    //Do something with the vectors...

    MPI\_Finalize();

    cout << "Execution time = " << TIMER\_ELAPSED/1000000.0 << endl;

        return 0;

}

//

//  read\_omp.cpp

//  412project

//

//  Created by Todd Dick on 12/16.

//  Copyright © 2016 Todd Dick. All rights reserved.

//

/\* This program will read in 10 .png files and write the data byte by byte to separate vector data type containers to be used for whatever purposes. This program is designed to be a baseline program to measure the time it takes to read in 10 media files. This program can be extended to do something with the vectors, but that is outside of the scope of this project.\*/

/\* There is no way to dynamically name vectors in C or C++, so this program will only use 10 files. To use more, more vectors need to be declared.\*/

/\* Prior to using this program, enter the following from command line in the directory containing your 10 media files. "ls \*.<filetype> > list.txt" without the quotes.\*/

/\* to compile --- g++ read\_omp.cpp -o omp -O3 -std=c++11 -fopenmp \*/

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstdint>

#include <vector>

#include <fstream>

#include <string>

#include <cstring>

#include <sys/time.h>

#ifdef \_OPENMP

#include <omp.h>

#endif

using namespace std;

#define TIMER\_CLEAR     (tv1.tv\_sec = tv1.tv\_usec = tv2.tv\_sec = tv2.tv\_usec = 0)

#define TIMER\_START     gettimeofday(&tv1, (struct timezone\*)0)

#define TIMER\_ELAPSED   ((long long) (tv3.tv\_usec)+((long long) (tv3.tv\_sec)\*1000000))

#define TIMER\_STOP      {gettimeofday(&tv2, (struct timezone\*)0);timersub(&tv2,&tv1,&tv3);}

struct timeval tv1,tv2,tv3;

vector<uint8\_t> file0;

vector<uint8\_t> file1;

vector<uint8\_t> file2;

vector<uint8\_t> file3;

vector<uint8\_t> file4;

vector<uint8\_t> file5;

vector<uint8\_t> file6;

vector<uint8\_t> file7;

vector<uint8\_t> file8;

vector<uint8\_t> file9;

void MakeList(vector<string> &list){

    ifstream infile;

    string line;

    infile.open("list.txt");

    if (!infile){

        cout << "error opening list.txt file" << endl;

        exit(1);

    }

    while(getline(infile,line)){

        list.push\_back(line);

    }

    infile.close();

}

// Hybrid of C/C++ methods for reading binary, because Jetson does not support C++14 and those are the methods I originally used.

void MakeVector(int number, vector<string> &list){

    FILE \* infile;

    string tempStr = list[number];

    const char\* filePath = tempStr.c\_str();

    infile = fopen(filePath,"rb");

    fseek(infile,0L,SEEK\_END);

    uint64\_t fileSize = ftell(infile);

    fseek(infile,0L,SEEK\_SET);

    uint8\_t buff[1] = {0};

    int i = 0;

    while( i < fileSize ){

        size\_t readStuff = fread(&buff,1,sizeof(uint8\_t),infile);

        switch(number){

            case 0:

                file0.push\_back(buff[0]);

                break;

            case 1:

                file1.push\_back(buff[0]);

                break;

            case 2:

                file2.push\_back(buff[0]);

                break;

            case 3:

                file3.push\_back(buff[0]);

                break;

            case 4:

                file4.push\_back(buff[0]);

                break;

            case 5:

                file5.push\_back(buff[0]);

                break;

            case 6:

                file6.push\_back(buff[0]);

                break;

            case 7:

                file7.push\_back(buff[0]);

                break;

            case 8:

                file8.push\_back(buff[0]);

                break;

            case 9:

                file9.push\_back(buff[0]);

                break;

            default:

                break;

        }

        i++;

    }

    fclose(infile);

}

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

    // generic useful variable(s)

    int i;

    // get a list of filenames

    vector<string> list;

    MakeList(list);

    // set up omp

    omp\_set\_dynamic(0);

    if(argc == 1){

        cout << "Usage : ./omp <number of threads>. Try again, set number of threads." << endl;

        exit(1);

    }

    int number = stoi(argv[1]);

    if(number > 10){

        cout << "Number of threads must be < 10." << endl;

        exit(1);

    }

    else{

        omp\_set\_num\_threads(number);

    }

    // start timer

    TIMER\_CLEAR;

    TIMER\_START;

#pragma omp parallel for

    for (i = 0; i < list.size(); i++){

        MakeVector(i,list);

    }

    TIMER\_STOP;

    cout << "Execution time = " << TIMER\_ELAPSED/1000000.0 << endl;

    //Do something with the vectors...

    return 0;

}

//

//  read\_pth.cpp

//  412project

//

//  Created by Todd Dick on 12/16.

//  Copyright © 2016 Todd Dick. All rights reserved.

//

/\* This program will read in 10 .png files and write the data byte by byte to separate vector data type containers to be used for whatever purposes. This program is designed to be a baseline program to measure the time it takes to read in 10 media files. This program can be extended to do something with the vectors, but that is outside of the scope of this project.\*/

/\* There is no way to dynamically name vectors in C or C++, so this program will only use 10 files. To use more, more vectors need to be declared.\*/

/\* Prior to using this program, enter the following from command line in the directory containing your 10 media files. "ls \*.<filetype> > list.txt" without the quotes.\*/

/\* to compile --- g++ read\_pth.cpp -o pth -O3 -std=c++11 -pthread\*/

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstdint>

#include <vector>

#include <fstream>

#include <string>

#include <cstring>

#include <sys/time.h>

#include <pthread.h>

using namespace std;

#define TIMER\_CLEAR     (tv1.tv\_sec = tv1.tv\_usec = tv2.tv\_sec = tv2.tv\_usec = 0)

#define TIMER\_START     gettimeofday(&tv1, (struct timezone\*)0)

#define TIMER\_ELAPSED   ((long long) (tv3.tv\_usec)+((long long) (tv3.tv\_sec)\*1000000))

#define TIMER\_STOP      {gettimeofday(&tv2, (struct timezone\*)0);timersub(&tv2,&tv1,&tv3);}

struct timeval tv1,tv2,tv3;

vector<uint8\_t> file0;

vector<uint8\_t> file1;

vector<uint8\_t> file2;

vector<uint8\_t> file3;

vector<uint8\_t> file4;

vector<uint8\_t> file5;

vector<uint8\_t> file6;

vector<uint8\_t> file7;

vector<uint8\_t> file8;

vector<uint8\_t> file9;

vector<string> globalList;

void MakeList(vector<string> &list){

    ifstream infile;

    string line;

    infile.open("list.txt");

    if (!infile){

        cout << "error opening list.txt file" << endl;

        exit(1);

    }

    while(getline(infile,line)){

        list.push\_back(line);

    }

    infile.close();

}

// Hybrid of C/C++ methods for reading binary, because Jetson does not support C++14 and those are the methods I originally used.

void MakeVector(int number){

    FILE \* infile;

    string tempStr = globalList[number];

    const char\* filePath = tempStr.c\_str();

    infile = fopen(filePath,"rb");

    fseek(infile,0L,SEEK\_END);

    uint64\_t fileSize = ftell(infile);

    fseek(infile,0L,SEEK\_SET);

    uint8\_t buff[1] = {0};

    int i = 0;

    while( i < fileSize ){

        size\_t readStuff = fread(&buff,1,sizeof(uint8\_t),infile);

        switch(number){

            case 0:

                file0.push\_back(buff[0]);

                break;

            case 1:

                file1.push\_back(buff[0]);

                break;

            case 2:

                file2.push\_back(buff[0]);

                break;

            case 3:

                file3.push\_back(buff[0]);

                break;

            case 4:

                file4.push\_back(buff[0]);

                break;

            case 5:

                file5.push\_back(buff[0]);

                break;

            case 6:

                file6.push\_back(buff[0]);

                break;

            case 7:

                file7.push\_back(buff[0]);

                break;

            case 8:

                file8.push\_back(buff[0]);

                break;

            case 9:

                file9.push\_back(buff[0]);

                break;

            default:

                break;

        }

        i++;

    }

    fclose(infile);

}

void \*ThreadFunc(void \*arg){

    int \*start\_end = (int \*) arg;

    int start = start\_end[0];

    int end = start\_end[1];

    for(int i = start; i < end; i++){

        MakeVector(i);

    }

    return arg;

}

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

    if (argc != 2) {

        cout << "Usage : ./pth <threadcount> ." << endl;

        exit(1);

    }

    vector<string> list;

    int i;

    MakeList(list);

    globalList = list;

    int thread\_count = stoi(argv[1]);

    TIMER\_CLEAR;

    TIMER\_START;

    int tid;

    pthread\_t threads[thread\_count];

    int start\_end[thread\_count][2];

    int errcode;

    int startVal = 0, endVal;

    int groupMod = 10 % thread\_count;

    for (tid = 0; tid < thread\_count; tid++){

        start\_end[tid][0] = startVal;

        int groupSize = 10/thread\_count;

        if (tid < groupMod){

            groupSize++;

        }

        start\_end[tid][1] = startVal+groupSize;

        startVal += groupSize;

        errcode = pthread\_create(&threads[tid], NULL, ThreadFunc,&start\_end[tid]);

        if (errcode) {

            cerr << "Pthread creation Error: " << strerror(errcode) << endl;

            exit(1);

        }

    }

    for ( tid = 0; tid < thread\_count; tid++) {

        errcode = pthread\_join(threads[tid], NULL);

    }

    if (errcode) {

        cerr << "Pthread join Error: " << strerror(errcode) << endl;

        exit(1);

    }

    TIMER\_STOP;

    cout << "Execution time = " << TIMER\_ELAPSED/1000000.0 << endl;

    //Do something with the vectors...

    return 0;

}