Dakota Gray & Todd Dick
CPE 435
04/27/17
Final Lab Project

Parallel and Serial FFT Implementation over UDP Server and Clients

Contents

Introduction

For the final project in CPE 435, each student was given the following assignment:
* Write two programs, Server and Client that communicate through UDP
* The Server must generate an array of 4096x4096 floating point numbers randomly
* The Server must divide the array into two sub arrays for use by two clients
* Each client (two clients) should create 4,8, or 16 threads to find the 1D-FFT for each row
* Each client must give its results back to the server
* All clients should work in parallel
* The server computes the 1D-FFT for each row serially
* The FFT results of the serial and parallel implementation should be matched
* The report should include timing comparisons of serial vs parallel with number of threads
* Any FFT implementation can be chosen

Todd Dick and Dakota Gray chose to work together on this project as undergraduates.

Overview

Project functional description

The FFT implementation used is radix-2 using the Cooley-Tukey algorithm. Cooley-Tukey is based on the divide-and-conquer method that breaks the DFT into smaller DFTS recursively. That this implementation is radix-2 means that the dimensions of the array being processed must be a power of 2. The specific algorithm for this FFT implementation makes use of the type complex (included in <complex> header) which allows the real and imaginary parts to be held from the FFT result. I will not go into detail on how the FFT is processed. Instead , we will describe the functionality of the server and client programs as a whole. The server program will setup a TCP socket for incoming connections from the clients. TCP was mistakenly used, due to the example c files posted to canvas for this project. It will then create the array of floating point numbers generated randomly; this array will be split into two sub-arrays and given to the clients to access. Next the server will handle incoming connections from clients over the TCP socket. At this time the client programs should be started. The clients will receive the names of the files that the sub-arrays were written to from the server. The server will now wait on the two clients to finish processing. The clients will prompt the user for a number of threads to use (4,8, or 16); then it will perform the FFT on its corresponding sub-array in parallel. The time taken to perform the parallel FFT calculation will be recorded by the client. After finishing calculation, the clients will give the server confirmation that they are done as well as the resulting FFT output. The clients will now close. Once receiving this confirmation, the server will calculate the FFT on its data and record the time taken to do so. Once finished the server will close.

Server

The server is run without user input or command line arguments. Upon starting, the server will create and bind to a TCP socket. Next the server will create a 4096 X 4096 array of type double; this array is filled with randomly generated floating point numbers (of size 0-1'000'000) one index at a time by use of two-level nested for loops. The server then splits this array of floating point numbers into two halves with two temporary arrays of size 2048 X 4096. These two sub-arrays are then written to files with the names “subArr1.dat” and “subArr2.dat” respectively. The memory used to hold the two sub-arrays is now freed. The original floating point array is now written into a new array of type complex double with the same dimensions and the memory from the original array is freed. The original floating point array was to write a file with plain floating point number values; the complex double array is what will actually be given to perform FFT on. The server now begins handling incoming connections from the clients. The first client connection accepted is treated as client 1 and the second is treated as client 2. Client 1 is sent the file name “subArr1.dat” over TCP; Client 2 is sent the file name “subArr2.dat” over TCP. These file names will tell the clients which sub-array they should perform calculations on. The server handles each of these client connections in child processes; the server will wait for those processes to finish before continuing. Once the clients are done the child processes will receive confirmation over TCP from the clients; at this point the child processes will exit. After seeing the children processes exit, the server will continue to calculate the FFT on its data serially. The FFT is handled with a series of methods; from main a single “FFT()” handler function is called with a pointer to a complex double array to be acted on, the dimensions of that array, and a step size. The FFT is performed 4096 times, once for each row. To do this, the FFT handler function is called 4096 times and given the pointer to each row of the 4096 X 4096 array. The dimension of each row is 4096, therefore the dimension given to the FFT is 4096. The step size used is 1.0; 1.0 was chosen to mimic the FFT results of Matlab. The time taken to perform the FFT here is recorded for analysis using methods and types given in <sys/time.h>. After performing the FFT calculations, the output (which is written into the complex double array) is written out to the file “fft_serial.dat”. Finally all dynamically allocated arrays are freed and the server is finished.

Client

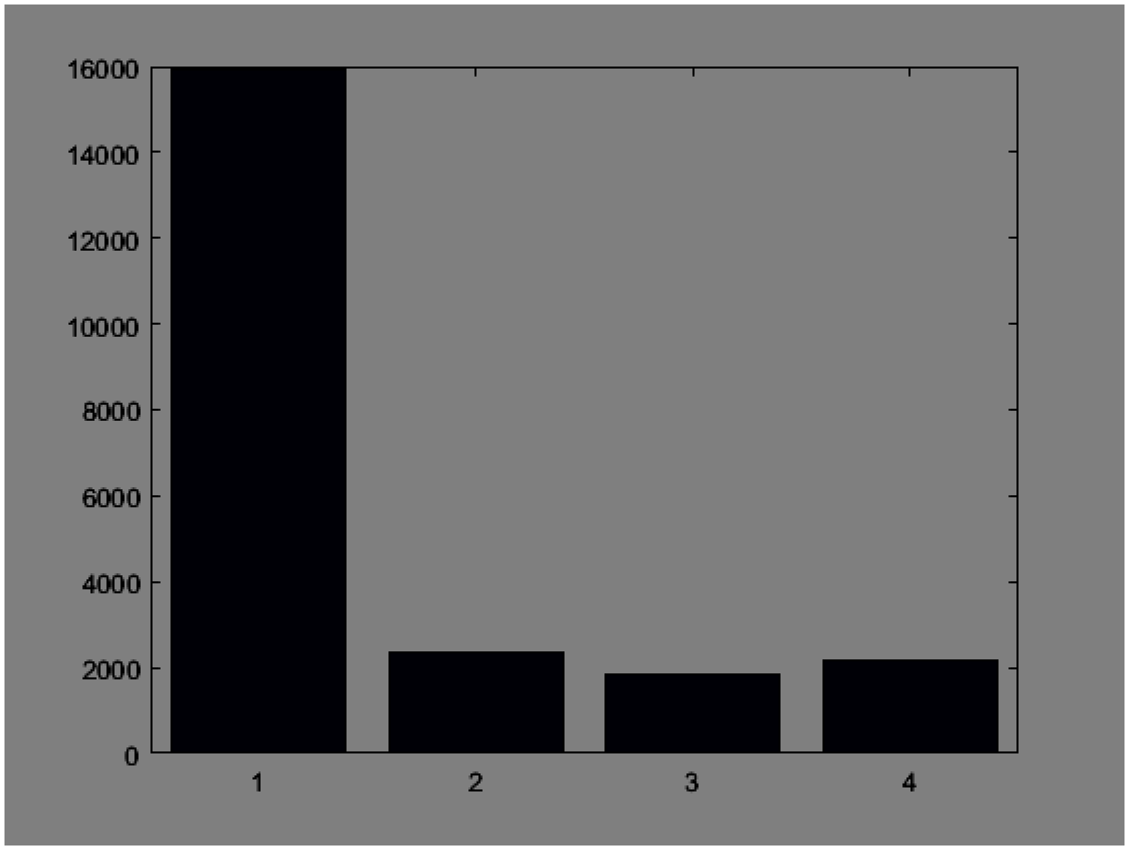
The client program should be started after the server program has already begun and created the TCP socket. The server produces standard output queues to alert the user visually of what is happening. Once started (assuming the server is running) the client will prepare to connect to the TCP socket and attempt to do so. Once connected the client will receive a string message over the socket and hold it in a character array buffer. This character buffer contains the file name of the file containing the sub-array this client will do work on. The client then determines which file name it received (“subArr1.dat” or “subArr2.dat”) and assigns itself as client 1 or client 2 by giving 1 or 2 to an integer named “cid”. The “cid” variable will be used in cases where the client must perform actions specifically after or before the other client. The client now opens the file it was assigned and prepares to read in its sub-array. A global array was declared in the client of type complex double pointer with length 2048. This global array will now be modified such that each of the 4096 pointers will point to dynamically allocated arrays of type complex double with length 4096. This allows for a globally accessible array of size 2048 X 4096 that can be used by the threads later on. Now the sub-array will be read in and stored in the 2048 X 4096 complex double array. Now the pthread environment variables can be created. First, the user is asked to input the number of threads (4,8, or 16). The number of rows from the complex double array that each thread will act on can now be calculated by dividing 2048 by the number of threads to be spawned. An integer array of size number of threads by 3 is created to hold argument values to be sent to each thread. This argument array will hold the thread id, the starting point of the array, and the ending point of the array. The start and end points will be used by the threads to determine what portion of the array they should be calling FFT() on. A for loop is used to create each thread; it will iterate a number of times equal to the number of threads to be created. Inside each iteration of the loop the argument array is modified to hold the start point, end point, and thread id for the thread about to be created. Next the thread is spawned to run a thread handling function called “threadFFT()” and given its 3 arguments from the argument array as parameters to the function. Once the threads are created, the root thread of client will wait for all threads to finish and “pthread_join”. Inside the thread handler function, first the argument values are extracted and stored; next, the size of the chunk of rows to be acted on is calculated by subtracting the start point from the end point. Now a for loop is created where the index will correspond to the row of the array to be operated on. The index will begin initialized to the start value given from the argument array. It will continue until the index reaches its end point. Within each iteration, FFT() is called and sent the global array index pointing to a 4096 length complex double array. The FFT() result is written to the row of the array it is acting on. The time taken from spawning the threads, until when they are all finished will be recorded and used for analyzing the time it takes to calculate FFT in parallel. Once the threads are finished, the client makes a system call to remove the files containing the sub-arrays; this is done to free up disk space as student disk space is limited to 1 Gigabyte. Next the client determined to be client 1 will write its FFT results to a file named “client_FFT.dat”; the client program determined to be client 2 will write its FFT results to the same file, after waiting to give time for client 1 to finish. After each client finishes writing its output to the files, it will send a message over UDP to the server telling the server it is done. When the server receives both messages it will know that the clients are finished with their output.

Timing Analysis

Timing analysis was performed on the server and client programs in the following manner: the time taken to compute the serial FFT on the server was recorded; the time taken for both clients to have their threads finish the jobs was recorded for 4, 8, and 16 threads. This was done 5 times for each case; the serial case was recorded each time, making 15 cases for the serial. The average will be calculated using these times. The individual times taken for clients 1 and 2 are recorded, however the largest of the two will be used as the time taken for each case. The recorded times are written as:

<client 1 time> | <client 2 time>

Run #	Serial (ms)	4 Thread (ms)	8 Thread (ms)	16 Thread (ms)
1	16305.8	2068.38 | 2570.71	-	-
2	15897.5	2072.26 | 1586.39	-	-
3	15840.2	2099.43 |2099.47	-	-
4	15917.4	2464.4 | 2431.47	-	-
5	15840.3	2132.27 | 2579.16	-	-
6	16180.2	-	1692.2 | 1980.78	-
7	15865.0	-	1702.0 | 1900.04	-
8	15865.2	-	1696.34 | 1890.92	-
9	16016.2	-	1684.49 | 1700.41	-
10	16069.0	-	1694.8 | 1714.24	-
11	15987.5	-	-	1901.74 | 2160.55
12	15848.2	-	-	1893.77 | 2216.73
13	15850.6	-	-	1864.29 | 2076.26
14	15851.2	-	-	1877.13 | 2104.18
15	15837.7	-	-	1907.1 | 2181.6
average	15944.8	2357.2	1837.278	2147.864



The above graph visually shows the average time differences between the serial, 4 thread, 8 thread, and 16 thread runs respectively.

Based on the results above, the average serial run time was 15.944.8 ms. The average run time for two clients in parallel running 4 threads was 2357.2 ms. The average run time for two clients in parallel running 8 threads was 1837.278 ms. The average rune time for two clients in parallel running 16 threads was 2147.864 ms. Below is a table listing the improvement based on the serial/parallel run time ratios.

Serial vs 4 thread clients	Serial vs 8 thread clients	Serial vs 16 thread clients
6.764	8.678	7.424

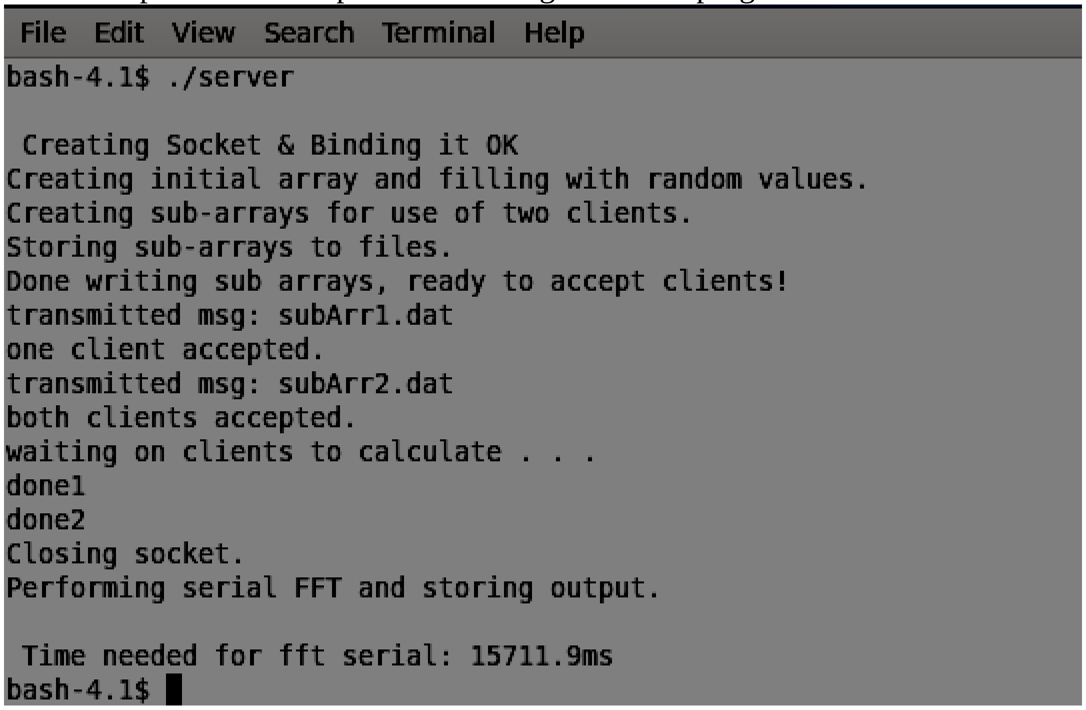
It is important to note that the clients were each operating on a problem set that was half the size of the problem set that the serial implementation worked on. There fore the speedup shown above is approximately twice that of what it would be on the entire problem set with the same number of threads. Based on the results, it appears that the 8 thread implementation was the fastest. The 16 thread implementation is likely slower due to the overhead of creating each of the additional threads and waiting on their completion.

Conclusion

In total over 25 hours were spent on this project. There were a number of challenges to overcome. Initially finding an FFT algorithm to implement took some time. Once the FFT algorithm was confirmed to work, setting up the TCP socket was a matter of following examples of previous labs and the files provided on Canvas. The next hurdle was to determine how to synchronize the work between the server and the client. We decided to use files as a means of storing and transferring data between. This worked initially, however we soon realized that the maximum disk quota given to students would be exceeded using this method. To fix this the files written out that were no longer needed are removed with system calls within the code. Implementing the parallelization was simple with using examples from previous labs as well as examples from CPE 412, Intro to Parallelization. Overall this project gave lengthy learning experience and solidified a lot of the skills learns within the previous labs.

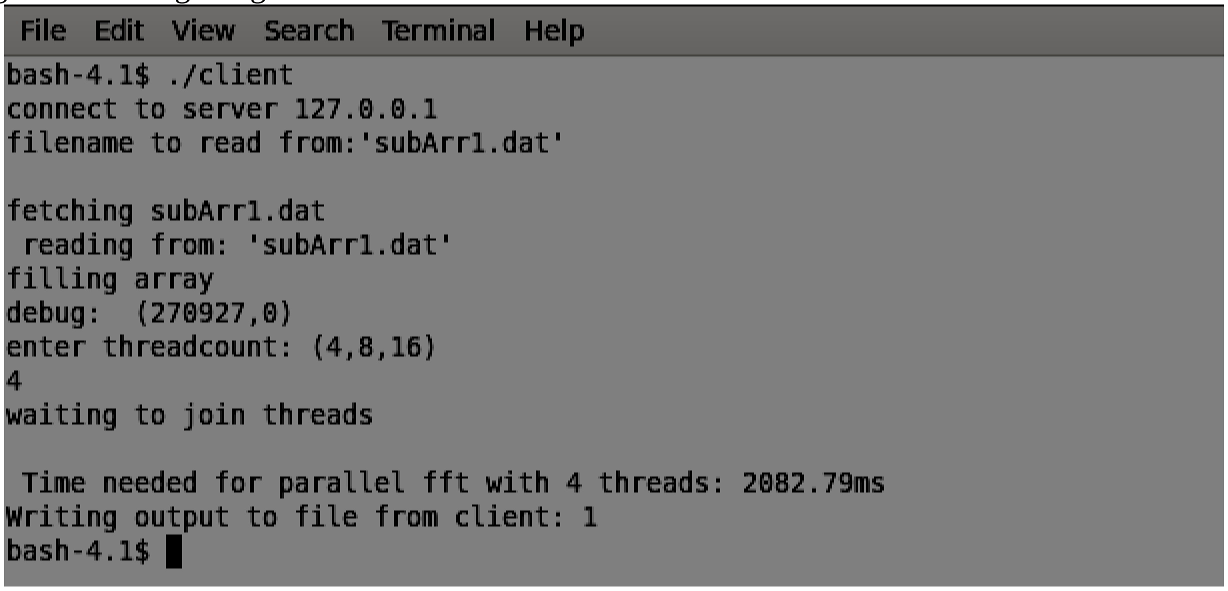
Appendix

Below is a screen capture of the output from running the server program:



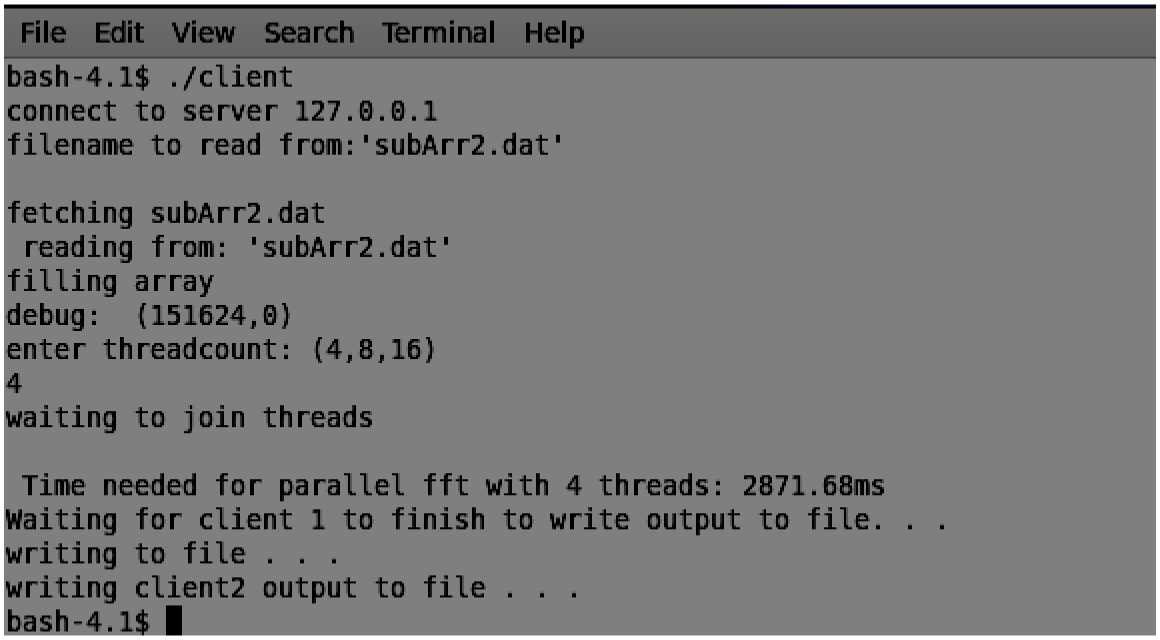
```
bash-4.1$ ./server

 Creating Socket & Binding it OK
Creating initial array and filling with random values.
Creating sub-arrays for use of two clients.
Storing sub-arrays to files.
Done writing sub arrays, ready to accept clients!
transmitted msg: subArr1.dat
one client accepted.
transmitted msg: subArr2.dat
both clients accepted.
waiting on clients to calculate . . .
done1
done2
Closing socket.
Performing serial FFT and storing output.

 Time needed for fft serial: 15711.9ms
bash-4.1$
```

Below is a screen capture of the output from running the client program and being designated as the first client:



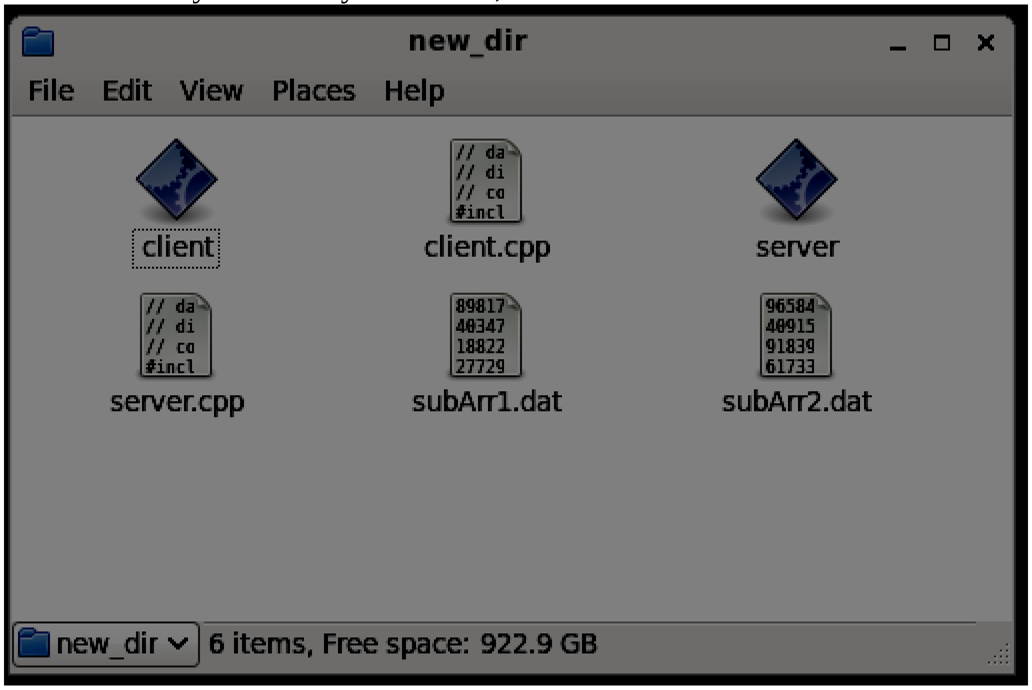
```
bash-4.1$ ./client
connect to server 127.0.0.1
filename to read from:'subArr1.dat'

fetching subArr1.dat
 reading from: 'subArr1.dat'
filling array
debug:  (270927,0)
enter threadcount: (4,8,16)
4
waiting to join threads

 Time needed for parallel fft with 4 threads: 2082.79ms
Writing output to file from client: 1
bash-4.1$
```

Below is a screen capture of the output from running the client program and being designated as the second client:



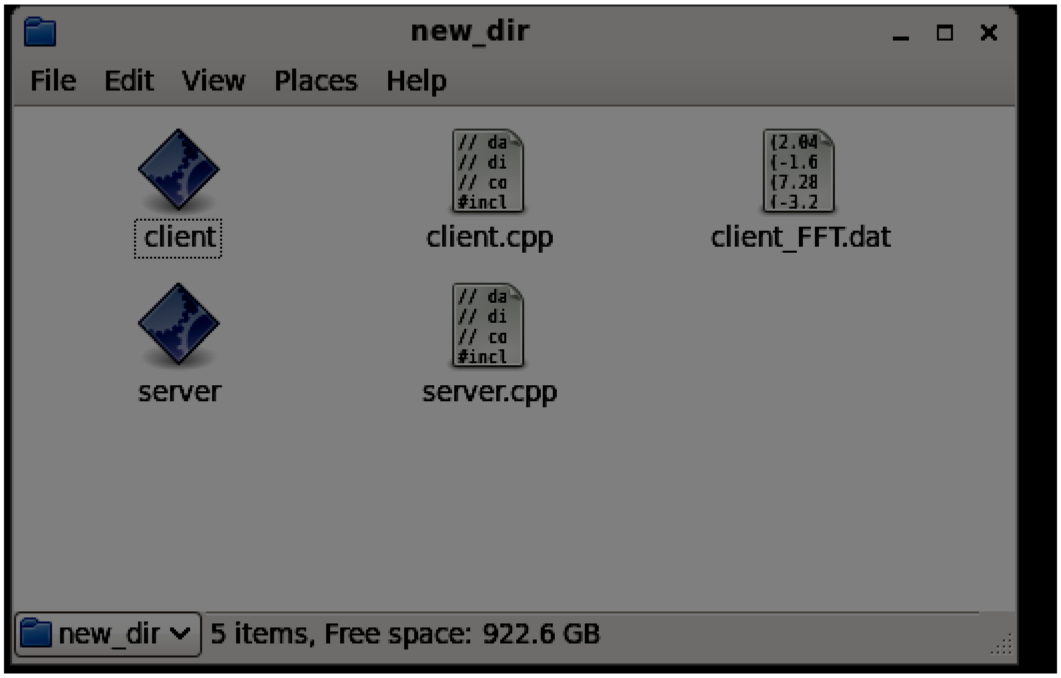
```
File  Edit  View  Search  Terminal  Help
bash-4.1$ ./client
connect to server 127.0.0.1
filename to read from:'subArr2.dat'

fetching subArr2.dat
 reading from: 'subArr2.dat'
filling array
debug:  (151624,0)
enter threadcount: (4,8,16)
4
waiting to join threads

 Time needed for parallel fft with 4 threads: 2871.68ms
Waiting for client 1 to finish to write output to file. . .
writing to file . . .
writing client2 output to file . . .
bash-4.1$
```

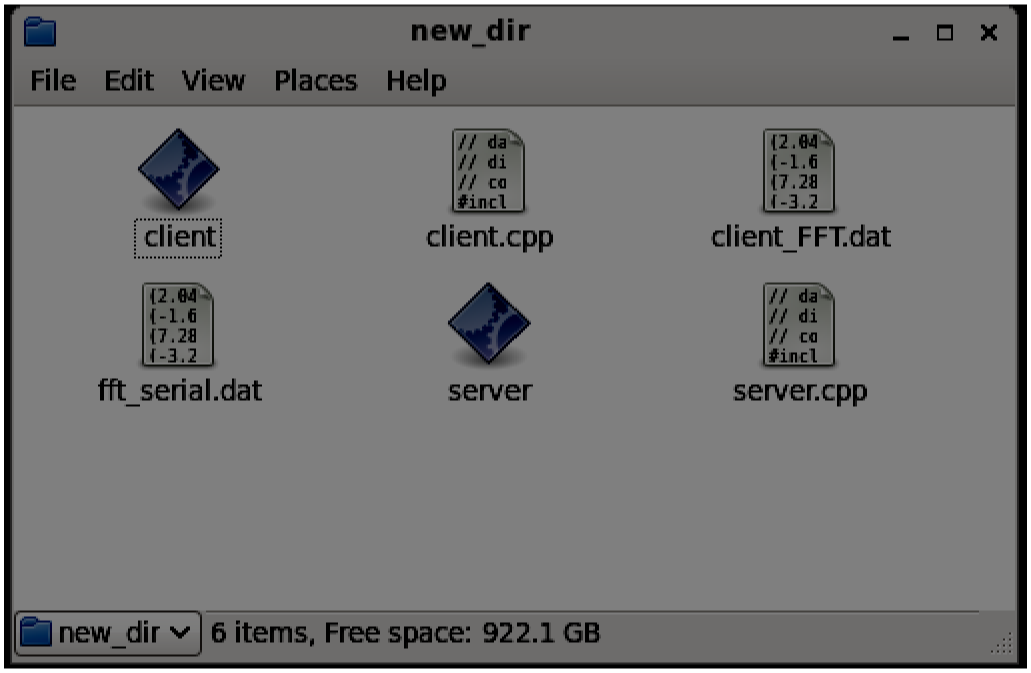
The server creates two sub arrays for use by the clients, this is shown below as a view from the directory:



The clients read in these sub array files to memory, and then delete the sub array files before writing their output to an output file. This is shown below as a view from the directory:

The server will perform its serial FFT after the clients, and write its output to its own file. The result of this is shown below as a view from the directory:


new_dir
File Edit View Places Help
client
client.cpp
client_FFT.dat
server
server.cpp
new_dir
5 items, Free space: 922.6 GB


new_dir
File Edit View Places Help
client
client.cpp
client_FFT.dat
fft_serial.dat
server
server.cpp
new_dir
6 items, Free space: 922.1 GB