

The purpose of this project was to write a distributed application using the Remote Procedure Call (RPC) protocol, the threads library, and the CUDA toolkit. In this project we were expected to use the ONC RPC Protocol to distribute computation via threads from a Linux workstation to two Linux workstations with the lowest two loads.

This project has the following files:

- *ldshr.x*
- *ldshr.c*
- *ldshr\_svc\_proc.c*
- *reduction.cu*
- *makefile*
- *datafile*

This project is divided into two parts. The client program running on a workstation provides the interface to the users, and that can be any workstation from our “**LINUX Lab**”. The servers running on the pre-selected five Linux workstations provide the machine load reporting service and another two computational services. We are instructed to use these workstations as the server workstations: “*arthur*”, “*bach*”, “*brahms*”, “*chopin*”, and “*degas*”.

Let’s begin with the 1<sup>st</sup> file, *ldshr.x*.

The code is a definition file for an RPC (Remote Procedure Call) interface, typically used with the ONC (Open Network Computing) RPC protocol. This is often described using a language specific to defining interfaces for RPC which likely uses the syntax for Sun Microsystems' RPC language, which can be compiled by *rpcgen* on Unix-like systems.

Here’s a breakdown of the components of the code:

### Struct Definitions:

- **input Struct:** This structure is designed to hold parameters needed by one of the RPC functions. It includes:
  - *N*: Likely the size of the array or some numeric parameter indicating the scale or number of elements.
  - *M*: Could represent a mean value used for calculations, possibly related to initializing values or determining behaviour of computation.
  - *S*: These are seed values, probably used for generating random numbers in a controlled way to ensure reproducibility.
- **Node Struct:** Represents a node in a linked list, which includes:
  - *F*: This could be a floating-point number representing data stored in each node.
  - *next*: A pointer to the next node in the linked list, establishing the linked list structure.
- **LinkedList Struct:** Represents a simple linked list structure that holds:
  - *head*: A pointer to the first node of the list.

## RPC Program Definition:

The RPC program is defined with a unique identifier and a version number. This definition specifies the procedures that can be remotely invoked, their signatures, and the types of data they accept and return.

- **program LDSHRPROG** :Declares an RPC program with a specific program number (**0X22200001**). This hexadecimal number uniquely identifies this particular set of RPC services on the network.
- **vesrion LDSHRVERS**:
  - **double 'getload(string)'** : This function likely retrieves and returns the load average of a system identified by a string (probably a hostname or an identifier). The load average could relate to CPU or other system resource metrics.
  - **double 'sumqroot\_gpu(input)'**: Computes and returns a double precision result, taking an input struct that contains parameters possibly configuring how the computation should be done on a GPU, like handling arrays using seeds for random number generation.
  - **double 'sumqroot\_lst(LinkedList)'** Similar to **'sumqroot\_gpu()'**, but it operates on a linked list, possibly performing some aggregation or transformation (like summing quadruple roots) on list elements.
- **1 = 1**: This line indicates the version of the program is **1**. It is crucial as different versions of an RPC program can have different APIs or behaviours.

## Use and Context:

This file is used to generate client and server stubs for RPC communication. The stubs handle data serialization and communication tasks, allowing developers to focus on implementing business logic rather than dealing with the complexities of network programming.

- **Client-side**: The client application uses these stubs to call functions as if they were local, but the calls are transparently sent over the network.
- **Server-side**: The server implements these functions, and the server stubs handle incoming network calls, execute the server's implementations, and send back the results.

These stubs and skeleton codes are generated using a tool like **rpcgen** by providing this **.x** file as input. This tool facilitates building distributed applications where servers can offer specific services that remote clients can invoke over a network.

The client program keeps a list of five available Linux workstations in our lab. It firstly invokes a remote procedure *'getloadO'* to collect the load average (over the last 5 minute) from each workstation in the list and selects two workstations with the lowest load averages. Then, based on the command line option (*'-gpu'* or *'-lst'*), the client creates two threads to invoke the computation function *'sumqroot\_gpuO'* or *'sumqroot\_lstO'* on the selected two workstations. Each workstation will get equal amount of data size. Finally, the client joins with the threads, adds the results from the two servers and prints it out.

"haydn % Idshr - gpu 20 5 17 23"

*"(executed on chopin and degas)"*

The first example, with the command line arguments `'-gpu NM S1 S2'`, computes the sum of the quadruple root of an array. The whole array has `'2N'` (i.e. `'220'`) elements. Each of the selected servers will initialize half of the array elements using the exponential distribution with the mean value `'M'` (i.e. `5`). The two servers will use the values `'S1'` and `'S2'`, respectively (i.e. `17` and `23`), as the initial seed. The computation is executed on the machines chopin and degas with their GPUs. The client combines the results from the two servers and prints it out.

```
"haydn % Idshr -lst datafile"
```

"(executed on arthur and degas)"

```
"result: 7.40"
```

The second example calculates the sum of the quadruple root of the doubles **5.0**, **7.0**, **19.0**, and **23.0** stored in the file `datafile`. The task is distributed to the machines "**arthur**" and "**degas**". Each server gets a linked list of doubles from the client.

The C code is the main client implementation for an **RPC**-based distributed computing system. It uses threads to offload computation to remote servers using different data inputs, either from an array or a linked list structure. The program is structured to handle different commands specified via command-line arguments.

Breakdown of the Code:

### Included Headers:

- "**stdio.h**" - Standard input/output library for functions like **printf**, **fscanf**, etc.
- "**ctype.h**" - Character type library, not directly used in the visible part of the code but could be included for character testing and conversion utilities.
- "**rpc/rpc.h**" - Contains functions and definitions for RPC programming, crucial for creating clients and making remote procedure calls.
- "**string.h**" - Provides string handling functions, likely used for operations like **strcmp**.
- "**pthread.h**" - Provides access to the POSIX threading API, enabling the program to create and manage threads.
- "**ldshr.h**" - Presumably a custom header related to the specific **RPC** service, defining data structures like **LinkedList**, **Node**, and **input**, as well as function prototypes.
- "**float.h**" - Provides limits for float types, used here to initialize minimum comparison values using **DBL\_MAX**.

### Struct Definitions:

- "**ThreadParams**"
  - Holds parameters for threads performing **GPU** computations.
  - Contains pointers to input structures and **CLIENT** for **RPC**.
- "**ListThreadParams**"
  - Used for threads processing linked lists.
  - Holds pointers to a **LinkedList** and an associated **CLIENT** for **RPC** operations.

### Function Descriptions:

- "**thread\_function**"
  - **Purpose:** Acts as a wrapper to call the '**sumqroot\_gpu\_10**' **RPC** function, handling **GPU** computations on remote servers.
  - **Parameters:** Takes a generic pointer, casts it to "**ThreadParams**", and uses it to perform an **RPC** call.
  - **Return:** Returns the result from the **RPC** call, or **NULL** if the call fails.
- "**readDataAndCreateLists**"
  - **Purpose:** Reads double values from a file and alternately populates two linked lists.
  - **Parameters:** Filename and pointers to two linked lists.
  - **Operations:** Opens the file, reads doubles, and allocates **Node** structures filled with read values, appending them alternately to the two lists.

- **Error Handling:** Checks for file opening errors and memory allocation failures, exiting on failure.
- **"processListfunction"**
  - **Purpose:** Processes a linked list using the '*sumqroot\_lst\_10*' *RPC* function.
  - **Parameters:** Takes "*ListThreadParams*" to access a linked list and its associated *RPC* client.
  - **Return:** Returns a pointer to a double holding the *RPC* call result or *NULL* if the call or memory allocation fails.
- **"main"**
  - **Initialization**
    - Parses command-line arguments to configure the computation.
    - Initializes *RPC* clients for pre-defined servers.
  - **Load Calculation**
    - Calls '*getload\_10*' for each server to determine their loads and selects the two with the lowest loads for computation tasks.
  - **Command Handling**
    - '*-lst*' option: Initializes linked lists, reads data into them, creates threads for processing these lists on the selected servers, and aggregates results.
    - '*-gpu*' option: Prepares input structures, creates threads for *GPU* computations on selected servers using the provided seed values, and aggregates results.
  - **Cleanup**
    - Frees allocated resources, destroys *RPC* clients, and handles potential thread-related results.

### Error Handling:

Throughout the program, extensive error handling ensures robust operation, including checking for:

- Successful memory allocations.
- Successful file operations.
- Valid *RPC* client creations.
- Correct execution of *RPC* calls.

This code is a sophisticated example of how multi-threading, remote procedure calls, and file I/O can be integrated in a *C* program to distribute computation tasks across multiple servers, making efficient use of networked resources and parallel processing.

## Code of *ldshr.c*:

```
EXPLORER
MARADHES [SSH: SPIRIT.EECS.CSUOHIO...]
CIS620
  Projects
    P1
    P2
    P3
    P4
      E datfile
      E ldshr
      C ldshr.cnt.c
      E ldshr.cnt.o
      E ldshr_svc
      C ldshr_svc_proc.c
      E ldshr_svc_proc.o
      C ldshr_svc.o
      E ldshr_vdr.c
      E ldshr_vdr.o
      C ldshr.c
      E ldshr.h
      E ldshr.o
      E ldshr.x
      M makefile
      C reduction.cu
      E reduction.o
      > test
      P4.zip
      > Desktop
      > Documents
      > Downloads
      > Music
      > Pictures
      > Public
      > screenshots
      > Templates
      > Videos
      $ bash_aliases
      E bash_aliases.save
      E bash_history
      $ bash_logout
      $ bashrc
      > OUTLINE
      > TIMELINE
  CIS620
    Projects
      P1
      P2
      P3
      P4
        E datfile
        E ldshr
        C ldshr.cnt.c
        E ldshr.cnt.o
        E ldshr_svc
        C ldshr_svc_proc.c
        E ldshr_svc_proc.o
        C ldshr_svc.o
        E ldshr_vdr.c
        E ldshr_vdr.o
        C ldshr.c
        E ldshr.h
        E ldshr.o
        E ldshr.x
        M makefile
        C reduction.cu
        E reduction.o
        > test
        P4.zip
        > Desktop
        > Documents
        > Downloads
        > Music
        > Pictures
        > Public
        > screenshots
        > Templates
        > Videos
        $ bash_aliases
        E bash_aliases.save
        E bash_history
        $ bash_logout
        $ bashrc
        > OUTLINE
        > TIMELINE
  CIS620
    Projects
      P1
      P2
      P3
      P4
        E datfile
        E ldshr
        C ldshr.cnt.c
        E ldshr.cnt.o
        E ldshr_svc
        C ldshr_svc_proc.c
        E ldshr_svc_proc.o
        C ldshr_svc.o
        E ldshr_vdr.c
        E ldshr_vdr.o
        C ldshr.c
        E ldshr.h
        E ldshr.o
        E ldshr.x
        M makefile
        C reduction.cu
        E reduction.o
        > test
        P4.zip
        > Desktop
        > Documents
        > Downloads
        > Music
        > Pictures
        > Public
        > screenshots
        > Templates
        > Videos
        $ bash_aliases
        E bash_aliases.save
        E bash_history
        $ bash_logout
        $ bashrc
        > OUTLINE
        > TIMELINE

1 #include <stdio.h>
2 #include <type.h>
3 #include <rpc/rpc.h>
4 #include <string.h>
5 #include "ldshr.h"
6 #include <pthread.h>
7 #include <float.h>
8
9 struct ThreadParams{
10     input_nms;
11     CLIENT *server;
12 };//traverse the pthread
13
14 typedef struct {
15     LinkedList *list;
16     CLIENT *server;
17 } ListThreadParams;
18 // Linked List into the pthread
19
20 void *thread_function(void *arg) {
21     struct ThreadParams *thread_params = (struct ThreadParams *)arg;
22     double *result = sumroot_gpu_1(thread_params->nms, thread_params->server);
23     if (result == NULL) {
24         fprintf(stderr, "Failed to get result from sumroot_1\n");
25         return NULL;
26     }
27     return result;
28 }
29
30 void readDataAndCreateLists(const char * filename, LinkedList *list1, LinkedList *list2) {
31     FILE *file = fopen(filename, "r");
32     if (file == NULL) {
33         perror("Failed to open file");
34         exit(EXIT_FAILURE);
35     }
36     double number;
37     int count = 0;
38     Node *currentNode;
39     while (fscanf(file, "%lf", &number) == 1) {
40         currentNode = malloc(sizeof(Node));
41         if (currentNode == NULL) {
42             fprintf(stderr, "Memory allocation failed\n");
43             exit(EXIT_FAILURE);
44         }
45         currentNode->f = number;
46         currentNode->next = NULL;
47         if (count % 2 == 0) {
48             currentNode->next = list1->head;
49             list1->head = currentNode;
50         } else {
51             currentNode->next = list2->head;
52             list2->head = currentNode;
53         }
54     }
55     count++;
56     fclose(file);
57 }
58
59 void *processListFunction(void *arg) {
60     ListThreadParams *params = (ListThreadParams *)arg;
61     double *result = sumroot_list_1(params->list, params->server);
62     if (result == NULL) {
63         fprintf(stderr, "Failed to get result from sumroot_list\n");
64         return NULL;
65     }
66     double *localResult = malloc(sizeof(double));
67     if (localResult == NULL) {
68         fprintf(stderr, "Failed to allocate memory for result\n");
69         return NULL;
70     }
71     *localResult = *result; // Copy the result to locally allocated memory to return
72     return localResult;
73 }
74
75 int main(argc, argv) int argc; char *argv[]; {
76     if (argc < 2) {
77         fprintf(stderr, "Usage: %s -option [additional args]\n", argv[0]);
78         exit(EXIT_FAILURE);
79     }
80     CLIENT *cl[5];
81     double *load[5];
82     char *server[5] = {"Arthur", "bach", "brahms", "chopin", "degas"};
83     // Create client handles
84     int i = 0;
85     while (i < 5) {
86         if (!cl[i] = clnt_create(server[i], LDHSHRPROC, LDHSHRVERS, "tcp")) {
87             fprintf(stderr, "Error creating client for server[%d]\n", i);
88             exit(1);
89         }
90         i++;
91     }
92     i = 0;
93     while (i < 5) {
94         load[i] = (double *)malloc(sizeof(double));
95         if (load[i] == NULL) {
96             fprintf(stderr, "Error allocating memory for load[%d]\n", i);
97             exit(1);
98         }
99         i++;
100     }
101     double min = DBL_MAX;
102     double second_min = DBL_MAX;
103     int min_index = -1;
104     int second_min_index = -1;
105     i = 0;
106     while (i < 5) {
107         *load[i] = *getload_1(&server[i], cl[i]);
108         if (*load[i] < min) {
109             second_min = min;
110             second_min_index = min_index;
111             min = *load[i];
112             min_index = i;
113         }
114         else if (*load[i] < second_min) {
115             second_min = *load[i];
116             second_min_index = i;
117         }
118         i++;
119     }
120     printf("Arthur: %lf bach: %lf brahms: %lf chopin: %lf degas: %lf\n", *load[0], *load[1], *load[2], *load[3], *load[4]);
121     printf("(executed on %s and %s)\n", server[min_index], server[second_min_index]);
122
123     if (strcmp(argv[1], "-list") == 0 && argc == 3) {
124         LinkedList *list1 = malloc(sizeof(LinkedList));
125         LinkedList *list2 = malloc(sizeof(LinkedList));
126         if (!list1 || !list2) {
127             fprintf(stderr, "Failed to allocate memory for Linked Lists\n");
128             exit(EXIT_FAILURE);
129         }
130     }
131 }
```

```

137 list1->head = list2->head = NULL;
138 readDataAndCreateLists(argv[2], list1, list2);
139
140 CLIENT *server1 = cl[min_index]; // More accurately reflects that this is a server connection
141 CLIENT *server2 = cl[second_min_index];
142
143 pthread_t threads[2];
144 ListThreadParams thread_params1 = {list1, server1};
145 ListThreadParams thread_params2 = {list2, server2};
146
147 pthread_create(&threads[0], NULL, processListFunction, &thread_params1);
148 pthread_create(&threads[1], NULL, processListFunction, &thread_params2);
149
150 double finalSum = 0.0;
151 void *status;
152 for (int i = 0; i < 2; i++) {
153     pthread_join(threads[i], &status);
154     if (status != NULL) {
155         finalSum += *(double *)status;
156     }
157 }
158
159 printf("Result: %.2f\n", finalSum);
160
161 // Cleanup
162 clnt_destroy(server1);
163 clnt_destroy(server2);
164 // Free linked lists here
165
166 }
167
168 else if (strcmp(argv[1], "-gpu") == 0) {
169     Click to collapse the range.
170     double finalSum = 0.0;
171     void *status;
172     pthread_t thread[2];
173     struct ThreadParams thread_params[2];
174
175     // nms[0].N = atoi(argv[2]) - 1;
176     // nms[0].M = atoi(argv[3]);
177     // nms[0].S1 = atoi(argv[4]);
178     // nms[1].N = atoi(argv[2]) - 1;
179     // nms[1].M = atoi(argv[3]);
180     // nms[1].S2 = atoi(argv[5]);
181     // thread_params[0].nms = &nms[0];
182     // thread_params[0].server = cl[min_index];
183     // thread_params[1].nms = &nms[1];
184     // thread_params[1].server = cl[second_min_index];
185     // pthread_create(&thread[0], NULL, thread_function, &thread_params[0]);
186     // pthread_create(&thread[1], NULL, thread_function, &thread_params[1]);
187
188     for (int i = 0; i < 2; i++) {
189         nms[i].N = atoi(argv[2]) - 1;
190         nms[i].M = atoi(argv[3]);
191         nms[i].S1 = atoi(argv[4]);
192         nms[i].S2 = atoi(argv[5]);
193         thread_params[i].nms = &nms[i];
194         thread_params[i].server = cl[i == 0 ? min_index : second_min_index];
195         pthread_create(&thread[i], NULL, thread_function, &thread_params[i]);
196     }
197
198     for (int i = 0; i < 2; i++) {
199         pthread_join(thread[i], &status);
200         if (status != NULL) {
201             double result = *(double *)status;
202             printf("Result from thread %d: %f\n", i, result);
203             finalSum += result;
204             //free(status); // Make sure to free the memory allocated in thread_function
205         } else {
206             fprintf(stderr, "Thread %d did not return a result\n", i);
207         }
208     }
209
210     printf("Result: %.2f\n", finalSum);
211
212     else {
213         fprintf(stderr, "Invalid option provided\n");
214         exit(EXIT_FAILURE);
215     }
216
217     return 0;
218 }
219
220 }
221
222 }

```

Now , the next file, **ldshr\_svc\_proc.c**:

The server provides three services (i.e. **functions**). To get the load average (*over the last 5 minute*) on Linux, you can call the system function '**getloadavg()**'. The function '**sumqroot\_gpu()**' firstly initializes the very large array using the values **N**, **M**, and **S1** (or **S2**) passed from the client. It then launches a kernel function on **GPU** to calculate the quadruple root of each element in the array. Next, it invokes the reduction kernel function on **GPU** to get the summation of the quadruple roots and returns the value back to the client. Similarly, the function '**sumqroot\_lst()**' gets a linked list of doubles from the client. It then utilizes higher-order functions, namely the '**map()**' function along with a formula function and the '**reduce()**' function along with a summation function, to compute the summation of the quadruple roots in the list.

The code implements a set of server-side **RPC** functions that perform computations on data structures and system information, using the **RPC** mechanism for distributed computing. Each function is designed to be called remotely by clients. Here's an explanation of each part:

## Included Headers:

- **"stdio.h"** - Used for input/output functions such as `printf` and ***fprintf***.
- **"string.h"** - Provides string manipulation capabilities, though its specific use isn't shown directly in the provided snippets.
- **"rpc/rpc.h"** - Necessary for functions and types used in ***RPC*** programming.
- **"ldshr.h"** - Presumably a custom header related to the specific ***RPC*** service, defining data structures like `LinkedList`, `Node`, and `input`, as well as function prototypes.
- **"unistd.h"** - Typically includes various constants, type definitions, and functions for performing system calls.
- **"math.h"** - Provides mathematical functions, in this case, `sqrt`, used for computing the quadruple root.

## Function Descriptions:

- **'getload\_1\_svcO'**
  - **Purpose:** Fetches and returns the system's 5-minute load average.
  - **Parameters:** ***char \*\*server*** for the server identifier (unused here) and ***struct svc\_req \*rqp*** for additional ***RPC*** request handling information.
  - **Process:** Uses **'getloadavgO'** to fetch load averages and returns the 5-minute average. It handles potential errors like a failed call or memory allocation issues.
  - **Return:** Pointer to a double containing the load average, or ***NULL*** on failure.
- **'sumqroot\_gpu\_1\_svcO'**
  - **Purpose:** Calculates a custom **"sum root"** computation, which could simulate a complex ***GPU***-based operation.
  - **Parameters:** ***input \*NMS***, containing parameters like array size, mean, and seeds; ***struct svc\_req \*rqp*** for the request.
  - **Process:** Chooses a seed based on the parity of ***N*** and performs a computation by calling **'sumqrootO'**, a function likely designed to simulate a computation-intensive task.
  - **Return:** Pointer to a double containing the computed value, or ***NULL*** if memory allocation fails.
- **'sumqroot\_lst\_1\_svcO'**
  - **Purpose:** Computes the sum of quadruple roots of all numbers in a linked list.
  - **Parameters:** ***LinkedList \*list*** which is the data to process, and ***struct svc\_req \*rqp*** for the request context.
  - **Process:** Iterates over the linked list, applying the map function to each node and accumulating results.
  - **Return:** Pointer to a static double containing the result. Static to ensure it remains valid post-function execution.
- **'map'**
  - **Purpose:** Computes the quadruple root of a number.
  - **Parameters:** ***Node \*node*** containing a double.
  - **Return:** The quadruple root of the node's value.
- **'reduce'**
  - **Purpose:** Accumulates the quadruple roots of all numbers in a linked list.
  - **Parameters:** ***LinkedList \*list*** containing the data.



- **Process:** Iterates through the list, applying map to each element and summing the results.
- **Return:** Sum of the mapped values.

## Server-Side Implementation Details:

These functions are typical of server-side ***RPC*** implementations where heavy lifting is done on the server, possibly to leverage more powerful computing resources or centralized data handling.

- The use of malloc in ***RPC*** service functions for returning data is crucial since ***RPC*** uses ***C's*** pass-by-value semantics, and returning local stack data would lead to undefined behaviour.
- Error handling is essential in such environments to ensure the server remains robust against failures caused by external factors like memory allocation failures or invalid inputs.

## Summary:

This server-side code effectively showcases how complex data operations can be abstracted behind ***RPC*** calls, allowing clients to perform potentially computationally intensive operations remotely. This pattern is beneficial in distributed systems where tasks can be offloaded to specialized or more powerful servers. The use of linked data structures and conditional logic based on input parameters demonstrates a flexible handling approach tailored to the needs of diverse client requests.

## Code of `ldshr_svc_proc.c`:

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <rpc/rpc.h>
4 #include "ldshr.h"
5 #include <unistd.h>
6 #include <math.h>
7
8 double sumroot(int N, int M, int S);
9 double map(Node *node);
10 double reduce(LinkedList *list);
11
12 double *getloadavg_1_svc(char **server, struct svc_req *rqstp) {
13     double load[3];
14     int ret = getloadavg(load, 3); // load[0], load[1], load[2] are 1, 5, and 15 minute load averages
15     if (ret == -1) {
16         fprintf(stderr, "getloadavg call failed\n");
17         return NULL; // You should handle the error appropriately
18     }
19     double *result = malloc(sizeof(double)); // Allocate memory for the result
20     if (result == NULL) {
21         fprintf(stderr, "Memory allocation failed\n");
22         return NULL;
23     }
24     *result = load[1]; // return the 5-minute load average, or choose another
25     return result;
26 }
27
28 double *sumroot_gpu_1_svc(input *NMS, struct svc_req *rqstp) {
29     // fprintf(stderr, "Error allocating memory for max in sumroot_gpu_1_svc\n");
30     double *max = (double *)malloc(sizeof(double));
31     *max = sumroot(NMS->N, NMS->M, NMS->S); // goes to cuda file
32     return max;
33 }
34
35 // double *sumroot_gpu_1_svc(input *NMS, struct svc_req *rqstp) {
36 //     double *max = (double *)malloc(sizeof(double));
37 //     if (max == NULL) {
38 //         fprintf(stderr, "Error allocating memory for max in sumroot_gpu_1_svc\n");
39 //         return NULL;
40 //     }
41 //     int seed = (NMS->N % 2 == 0) ? NMS->S1 : NMS->S2;
42 //     *max = sumroot(NMS->N, NMS->M, seed); // Using dynamic seed based on N
43 //     return max;
44 // }
45
46 double *sumroot_list_1_svc(LinkedList *list, struct svc_req *rqstp) {
47     static double result; // Make static to ensure it persists after function return
48     result = reduce(list);
49     return &result;
50 }

```

```

50 }
51
52 double map(Node *node) {
53     return sqrt(sqrt(node->f)); // Quadruple root
54 }
55
56 double reduce(Node **list) {
57     double sum = 0;
58     Node *current = list->head;
59     while (current != NULL) {
60         sum += map(current);
61         current = current->next;
62     }
63     return sum;
64 }

```

Result: 142129.93

Next, the file **reduction.cu**:

The **CUDA C++** program demonstrates a distributed computation approach using **GPU** acceleration to compute the sum of the quadruple roots of an array of numbers, structured in a template fashion with kernel functions for map and reduce phases. Here's a detailed explanation of each part of the code:

### Included Headers:

- **"stdio.h"** - Standard C library for input and output functions.
- **"stdlib.h"** - Standard C library for memory allocation, process control, conversions, and others.
- **"cuda\_runtime.h"** - **CUDA** runtime **API**, essential for using **CUDA** functionalities like memory allocation, data transfer, and kernel execution.

### Struct Definition:

- **"SharedMemory"**:  
This template struct provides a type-safe way to access shared memory within **CUDA** kernels. It uses the `extern` keyword to declare shared memory arrays (`__smem[]`) that are scoped to individual blocks of threads.
  - **operator \* T** and **operator const T \*** - These operator overloads allow instances of **SharedMemory < T >** to be used directly as pointers to type **T**, providing access to shared memory as if it were an array of type **T**.

### CUDA Kernels:

- **'map' Kernel:**  
Computes the quadruple root (fourth root) of each element in the input array if the element is non-negative.
  - **Parameters:**
    - **double ' \* g\_idata'** : Input data array.
    - **double ' \* g\_odata'** : Output data array where results are stored.
    - **unsigned int 'n'**: Number of elements in the input and output arrays.
- **'reduce' Kernel:**
  - Performs a parallel reduction to sum all elements in the array.
  - Uses shared memory for efficient intra-block summation.
  - Each block computes a partial sum of its elements, and the results are written to the output array.
  - **Shared Memory Usage:** Efficient for reduction as it minimizes global memory access.

### Helper Function: *gpuAssert*

A utility function for error checking *CUDA* operations. It prints out errors and optionally aborts the program.

- **Parameters:**
  - *cudaError\_t code*: The *CUDA* error code.
  - *const char \* file*: The file in which the error occurred.
  - *int line*: The line number at which the error is checked.
  - *bool abort*: If true, the function exits the program on an error.

### Extern "C" Function: *sumqroot*

The main computational function that sets up *GPU* memory, initiates kernel launches, and handles data transfer between host and device.

- **Parameters:**
  - *int 'N'*: Power of two that determines the number of elements in the computation.
  - *int 'M'*: Mean value used to generate the input data using an exponential distribution.
  - *int 'S'*: Seed for random number generation.
- **Process:**
  - Allocates host and device memory.
  - Initializes input data using an exponential distribution.
  - Calls the map kernel to compute quadruple roots.
  - Uses the reduce kernel in a loop to sum all elements, handling intermediate results with dynamic kernel launches as the number of elements reduces.
- **CUDA Memory Management:**
  - Uses *cudaMalloc* to allocate memory on the *GPU*.
  - Transfers data between host and device using *cudaMemcpy*.
  - Frees *GPU* memory resources with *cudaFree*.
- **Loop for Reduction:**

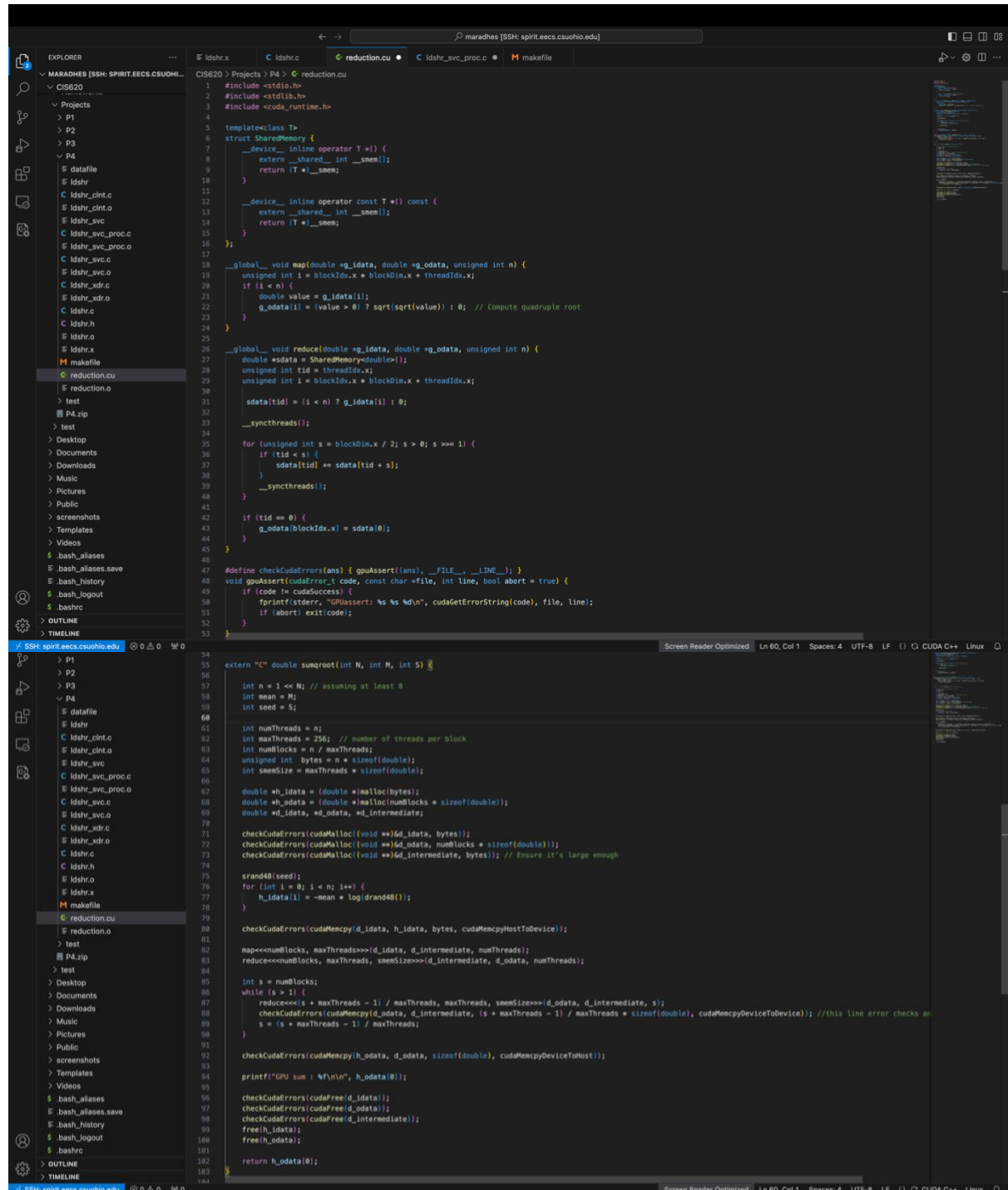
The reduction process may require multiple stages depending on the number of threads and blocks. The loop continues reducing the size of the data (s) until only one value remains, which is the sum of all quadruple roots.
- **Error Handling:**

Every critical *CUDA* operation is checked using *checkCudaErrors*, which uses *gpuAssert* to validate the operation's success.

### Summary

This program exemplifies a typical use case for *CUDA* in performing high-performance parallel computations, combining memory management, error handling, and parallel programming patterns to efficiently process large data sets on the *GPU*.

## Code of *reduction.cu*:



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cuda_runtime.h>
4
5 template<class T>
6 struct SharedMemory {
7     __device__ inline operator T*() {
8         extern __shared__ int __smem[];
9         return (T*)__smem;
10    }
11
12     __device__ inline operator const T*() const {
13         extern __shared__ int __smem[];
14         return (T*)__smem;
15    }
16 };
17
18 __global__ void map(double* g_idata, double* g_odata, unsigned int n) {
19     unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
20     if (i < n) {
21         double value = g_idata[i];
22         g_odata[i] = (value > 0) ? sqrt(sqrt(value)) : 0; // Compute quadruple root
23     }
24 }
25
26 __global__ void reduce(double* g_idata, double* g_odata, unsigned int n) {
27     double* sdata = SharedMemory<double>();
28     unsigned int tid = threadIdx.x;
29     unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
30
31     sdata[tid] = (i < n) ? g_idata[i] : 0;
32
33     __syncthreads();
34
35     for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
36         if (tid < s) {
37             sdata[tid] += sdata[tid + s];
38         }
39         __syncthreads();
40     }
41
42     if (tid == 0) {
43         g_odata[blockIdx.x] = sdata[0];
44     }
45 }
46
47 #define checkCudaErrors(ans) { gpuAssert((ans), __FILE__, __LINE__); }
48 void gpuAssert(cudaError_t code, const char* file, int line, bool abort = true) {
49     if (code != cudaSuccess) {
50         fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
51         if (abort) exit(code);
52     }
53 }
54
55 extern "C" double sumroot(int N, int M, int S) {
56     int n = 1 << N; // assuming at least 8
57     int mean = M;
58     int seed = S;
59
60     int numThreads = n;
61     int maxThreads = 256; // number of threads per block
62     int numBlocks = n / maxThreads;
63     unsigned int bytes = n * sizeof(double);
64     int smemSize = maxThreads * sizeof(double);
65
66     double* h_idata = (double*)malloc(bytes);
67     double* h_odata = (double*)malloc(numBlocks * sizeof(double));
68     double* d_idata, *d_odata, *d_intermediate;
69
70     checkCudaErrors(cudaMalloc((void**)&d_idata, bytes));
71     checkCudaErrors(cudaMalloc((void**)&d_odata, numBlocks * sizeof(double)));
72     checkCudaErrors(cudaMalloc((void**)&d_intermediate, bytes)); // Ensure it's large enough
73
74     srand48(seed);
75     for (int i = 0; i < n; i++) {
76         h_idata[i] = -mean * log(rand48());
77     }
78
79     checkCudaErrors(cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice));
80
81     map<<numBlocks, maxThreads>>>(d_idata, d_intermediate, numThreads);
82     reduce<<numBlocks, maxThreads, smemSize>>>(d_intermediate, d_odata, numThreads);
83
84     int s = numBlocks;
85     while (s > 1) {
86         reduce<<= maxThreads - 1 / maxThreads, maxThreads, smemSize>>>(d_odata, d_intermediate, s);
87         checkCudaErrors(cudaMemcpy(d_odata, d_intermediate, (s + maxThreads - 1) / maxThreads * sizeof(double), cudaMemcpyDeviceToDevice)); //this line error checks an
88         s = (s + maxThreads - 1) / maxThreads;
89     }
90
91     checkCudaErrors(cudaMemcpy(h_odata, d_odata, sizeof(double), cudaMemcpyDeviceToHost));
92
93     printf("GPU sum : %f\n", h_odata[0]);
94
95     checkCudaErrors(cudaFree(d_idata));
96     checkCudaErrors(cudaFree(d_odata));
97     checkCudaErrors(cudaFree(d_intermediate));
98     free(h_idata);
99     free(h_odata);
100
101     return h_odata[0];
102 }
```

Let's deal with the *datafile*:

This *datafile* is used as input for some kind of processing, likely a computation based on the context of previous messages. In the context of your distributed computing system, this file could be read by the *ldshr.c* client application.

In the *ldshr.c* file, when the program is run with the *'-lst'* command-line option, the main function will invoke the *"readDataAndCreateLists"* function to read numbers from the datafile and populate two linked lists with its contents.

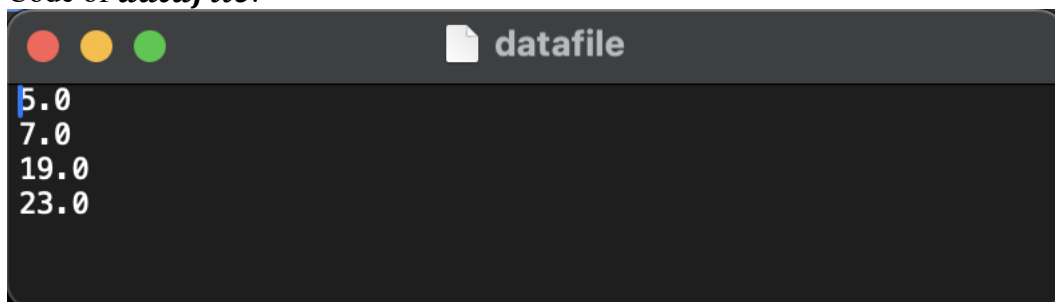
Here's a brief rundown of how that part of the program works:

- **Command-Line Argument Check:** The program first checks the provided arguments to see if the *'-lst'* option is specified.  

```
if (strcmp(argv[1], -lst) == 0 && argc == 3){ }
```
- **Opening the Data File:** If the *'-lst'* condition is met, *"readDataAndCreateLists"* is called with the filename provided as the second argument (here, it would be *argv[2]*, which is expected to be *"datafile"*).
- **Reading Data and Filling Lists:** The *"readDataAndCreateLists"* function then opens the datafile and reads its contents. For each number read from the file, a new Node is created, and the number is stored in this node. The nodes are alternately added to two linked lists (list1 and list2), effectively splitting the data between them.
- **Processing Lists:** After the lists are populated, two threads are created, each responsible for processing one of the lists. The *"processListfunction"* function is used as the entry point for each thread, which in turn calls the *'sumqroot\_lst\_1\_svc()'* *RPC* function to calculate the sum of quadruple roots of the numbers in each list.
- **Result Aggregation:** Once both threads have finished processing, the results (sum of quadruple roots from each list) are combined to produce a final result, which is then outputted by the program.

In summary, when the *ldshr.c* program is invoked with the *'-lst'* option followed by a filename, it reads from that file to create two linked lists of numbers. These lists are then processed in parallel using remote procedure calls, and the results are aggregated and printed out. The datafile in this case provides the raw data for this distributed computing operation.

Code of *datafile*:



```
5.0
7.0
19.0
23.0
```

Lastly, the file, **makefile**:

The **makefile** is a script used by the make utility to build and compile a distributed application that utilizes both CPU and GPU for processing. It defines a series of rules and dependencies to automate the compilation of a program named **ldshr**.

Let's go through it section by section:

### Variables:

- **cc**: The compiler used for **CUDA** files (**.cu**), typically **nvcc** (NVIDIA CUDA Compiler).
- **APPN** : The base name of the application, **ldshr** in this case.
- **LIBS** : Libraries to link with during compilation, here **-lpthread** for linking with the **POSIX** threading library.

### Targets and Dependencies:

- **all**: The default target. It depends on "**ldshr.h**", "**reduction.o**", **\$(APPN)\_svc**, and **\$(APPN)**. Running make without arguments will build all these dependencies.
- "**reduction.o**": This target compiles the **reduction.cu** file into an object file "**reduction.o**" using the **CUDA** compiler. This object file is likely to contain the **GPU**-accelerated functions.
- **\$(APPN).h** : Generates header files from an **RPC** interface definition file (**\$(APPN).x**) using **rpcgen**, a tool for generating **C** code to implement an **RPC** protocol.
- **\$(APPN)\_svc** : Compiles the server component of the **RPC** application. It links together several object files, including those for the server procedures, the server stub, the **XDR** routines (for data serialization), and the "**reduction.o**" file.
- **\$(APPN)\_svc.o** , **\$(APPN)\_xdr.o** , **\$(APPN)\_clnt.o** : These targets compile individual source files **\$(APPN)\_svc.c**, **\$(APPN)\_xdr.c**, **\$(APPN)\_clnt.c** into object files. Each source file is generated by **rpcgen** and handles different parts of the **RPC** protocol:
  - **\_svc.c** : Server stubs for handling **RPC** requests.
  - **\_xdr.c** : **XDR** routines for data serialization and deserialization.
  - **\_clnt.c** : Client stubs for making **RPC** calls to the server.
- **\$(APPN)** : Compiles the client executable. It depends on the client object files and **XDR** routines, and it uses **gcc** for linking because the client part is likely not containing any **CUDA** code, thus not requiring **nvcc**.

### Phony Target: clean

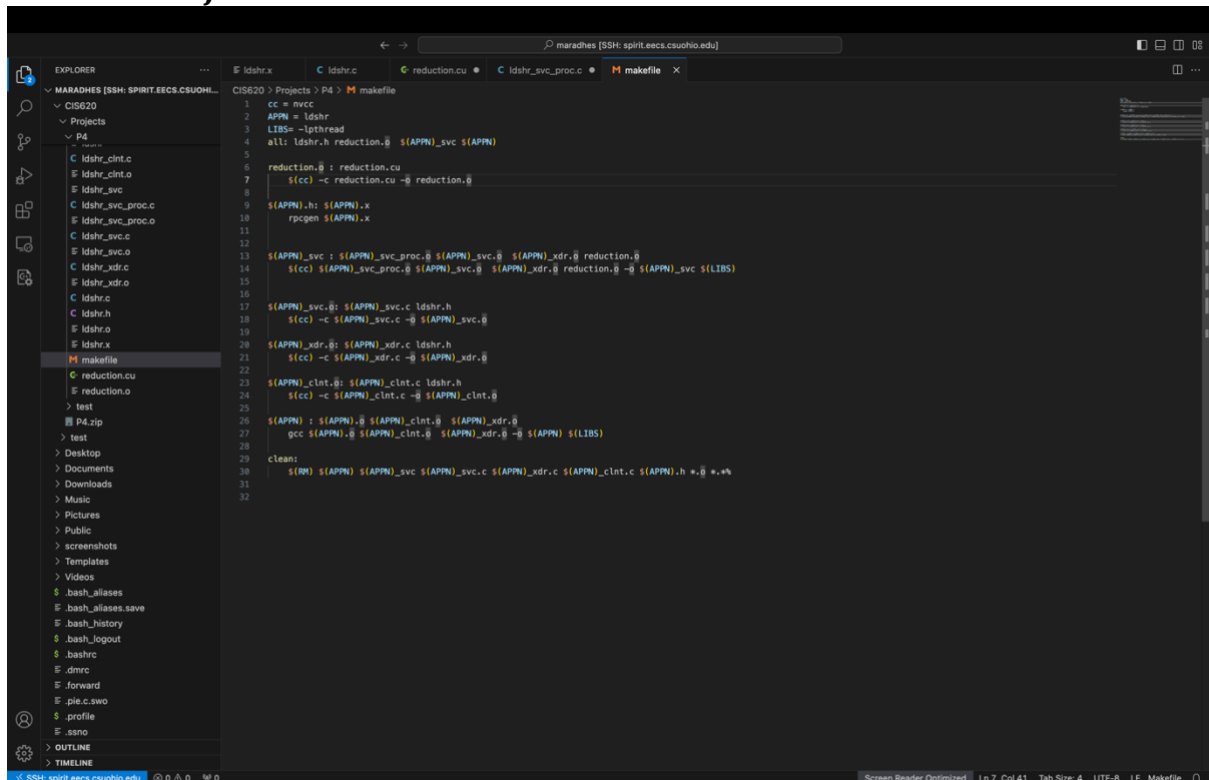
- **clean**: A phony target that doesn't correspond to a file name; it's a command for cleaning up the build environment. It removes the application executables, intermediate object files (**\*.o**), and files generated by **rpcgen** (with extensions **.c** and **.h**).

## Special Characters:

- $\$@$  : Represents the target name.
- $\$^$  : Represents all of the target's dependency names.
- $\$(RM)$  : The command to remove files, typically ***rm -f***.

In essence, this ***makefile*** provides a sequence of instructions for building a distributed application that includes both a client and a server. The server uses ***CUDA*** for computations, while the client and server communicate over ***RPC***. The ***makefile*** ensures that any changes to the source files trigger a rebuild of the affected components, facilitating a smooth development workflow.

## Code of ***makefile***:



```
1 cc = nvcc
2 APPN = ldshr
3 LIBS = -lpthread
4 all: ldshr.o reduction.o $(APPN)_svc $(APPN)
5
6 reduction.o: reduction.cu
7 $(cc) -c reduction.cu -o reduction.o
8
9 $(APPN).h: $(APPN).x
10 rpcgen $(APPN).x
11
12 $(APPN)_svc: $(APPN)_svc_proc.o $(APPN)_svc.o $(APPN)_xdr.o reduction.o
13 $(cc) $(APPN)_svc_proc.o $(APPN)_svc.o $(APPN)_xdr.o reduction.o -o $(APPN)_svc $(LIBS)
14
15 $(APPN)_svc.o: $(APPN)_svc.c ldshr.h
16 $(cc) -c $(APPN)_svc.c -o $(APPN)_svc.o
17
18 $(APPN)_xdr.o: $(APPN)_xdr.c ldshr.h
19 $(cc) -c $(APPN)_xdr.c -o $(APPN)_xdr.o
20
21 $(APPN)_clnt.o: $(APPN)_clnt.c ldshr.h
22 $(cc) -c $(APPN)_clnt.c -o $(APPN)_clnt.o
23
24 $(APPN) : $(APPN).o $(APPN)_clnt.o $(APPN)_xdr.o
25 gcc $(APPN).o $(APPN)_clnt.o $(APPN)_xdr.o -o $(APPN) $(LIBS)
26
27 clean:
28 $(RM) $(APPN) $(APPN)_svc $(APPN)_svc.c $(APPN)_xdr.c $(APPN)_clnt.c $(APPN).h *.o *.x
```

Project status: ***COMPLETELY WORKING.***

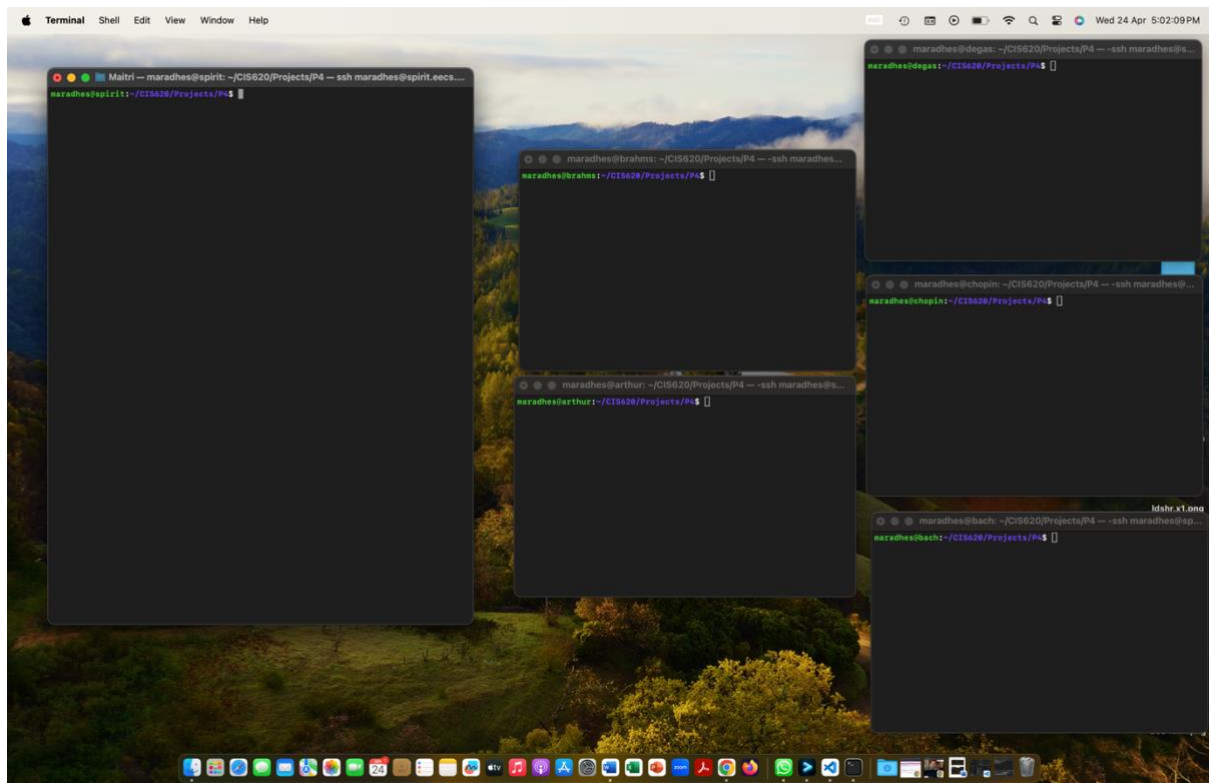
Testing and debugging:

We got issues with the client computing the sum to **0.00**. The issue was with the sum the results in the client program( ***ldshr.c***). The thread's result was incorrect and we fixed it by freeing the allocated memory and then correctly summing them up and printing out the total.

Outputs:

Step-1:

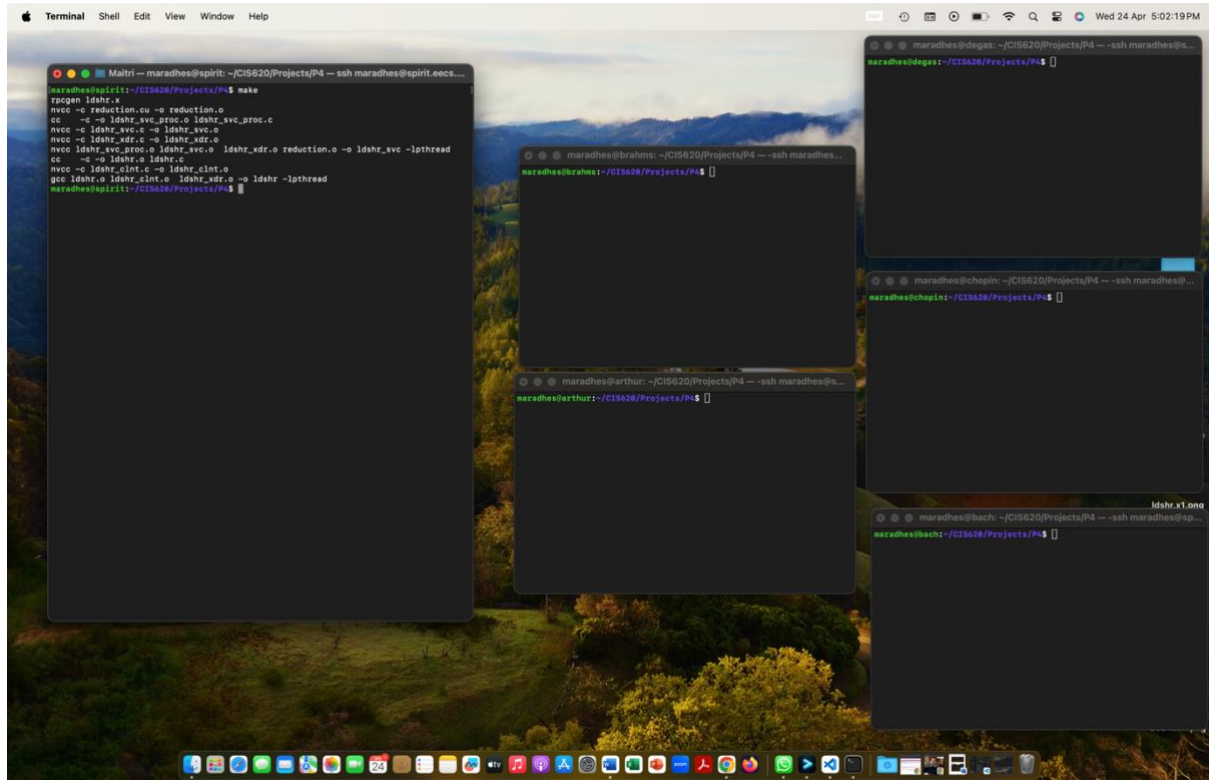
- To begin to run the code, we need to open 6 terminals in total.
  - 5 server terminals
  - 1 client terminal.





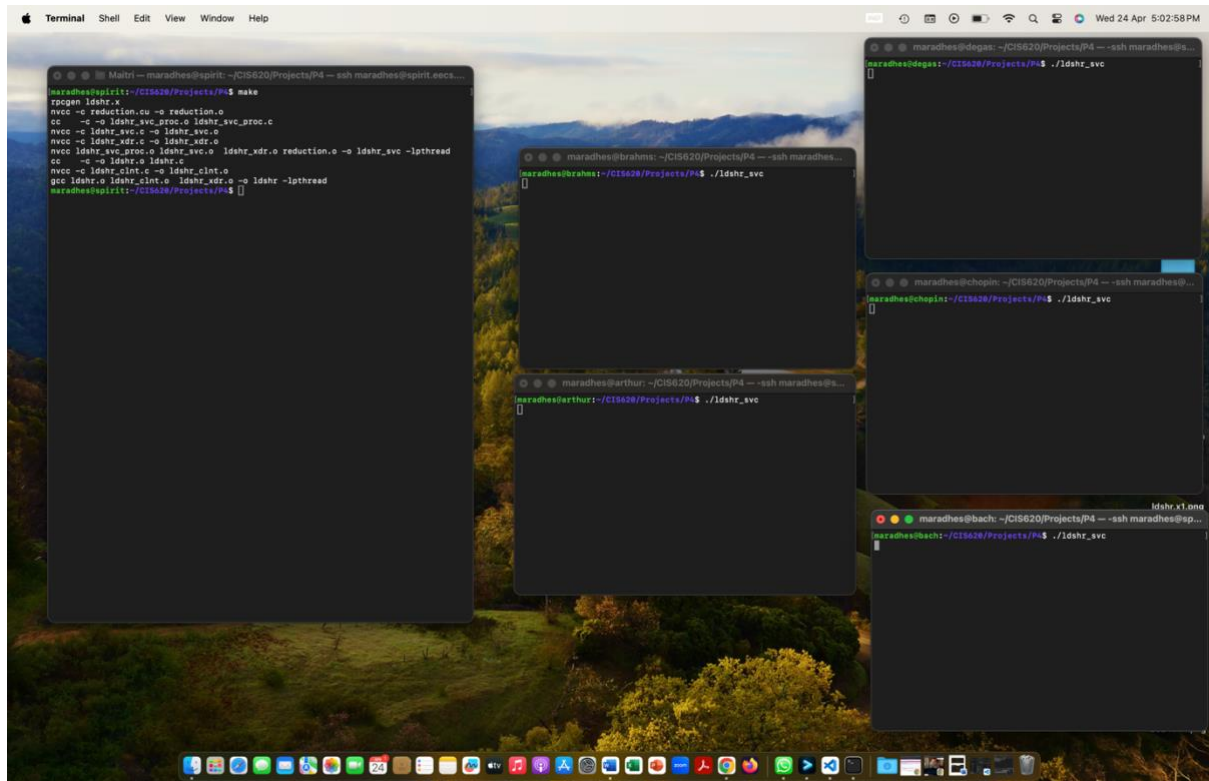
## Step 2:

- Make all the necessary files to compile the program. To do that, we make use of the **make** command. Inside our **makefile**, we also **rpcgen** here which is used to create the 4 **RPC** connected files. From these generated files, we import "**ldshr.h**" file into the **ldshr.c** and **ldshr\_svc\_proc.c** files as we want to make use of the "**ldshr.x**" file.
- Open five terminals in the specified hostnames specified in **ldshr.c**.
- The hardcoded hostnames are "**arthur**", "**bach**", "**brahms**", "**chopin**", and "**degas**".



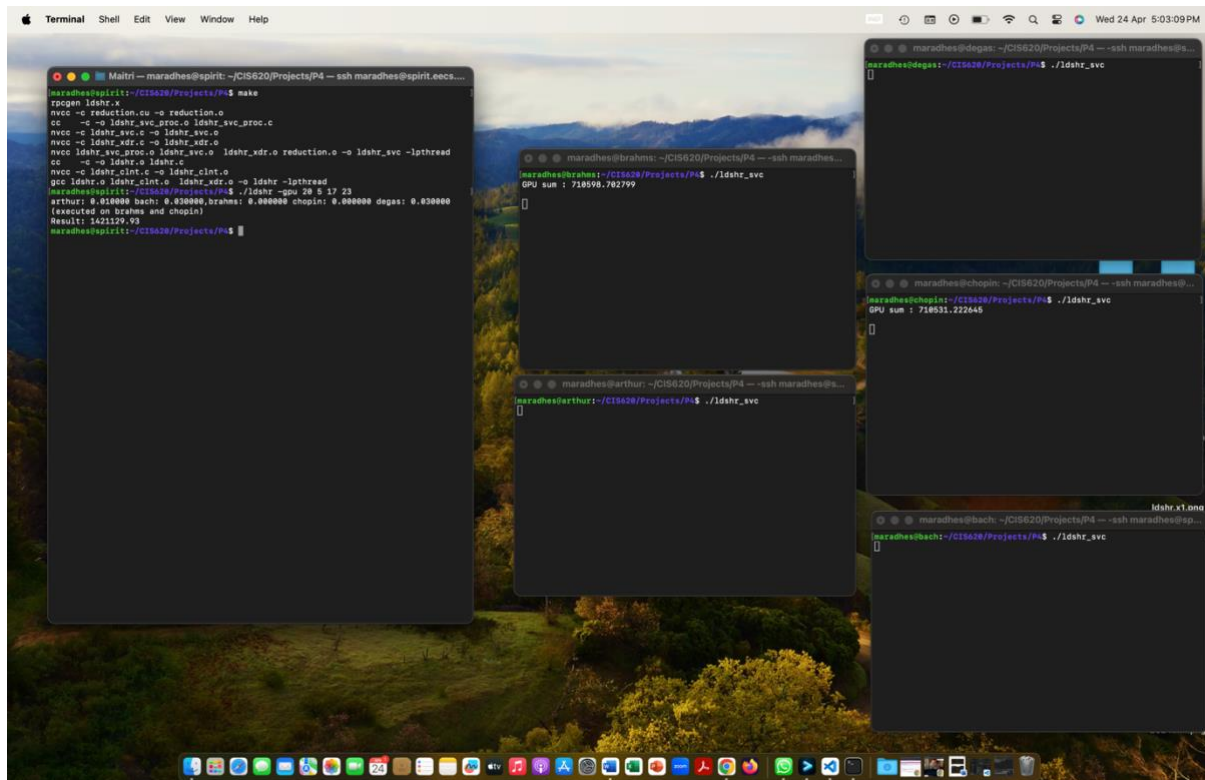
### Step 3:

- The servers are setup in all the specified host servers and run. The servers are run, and they are listening to incoming requests.
- We need to 1<sup>st</sup> run the **ldshr\_svc\_proc.c** file making use of the **./ldshr\_svc** command so that the servers are wanting to connect to the client workstation, **ldshr.c**.
- If you do not run the server file 1<sup>st</sup>, the client workstation will throw an error that it could not connect to the servers to get the load.



## CASE 1:

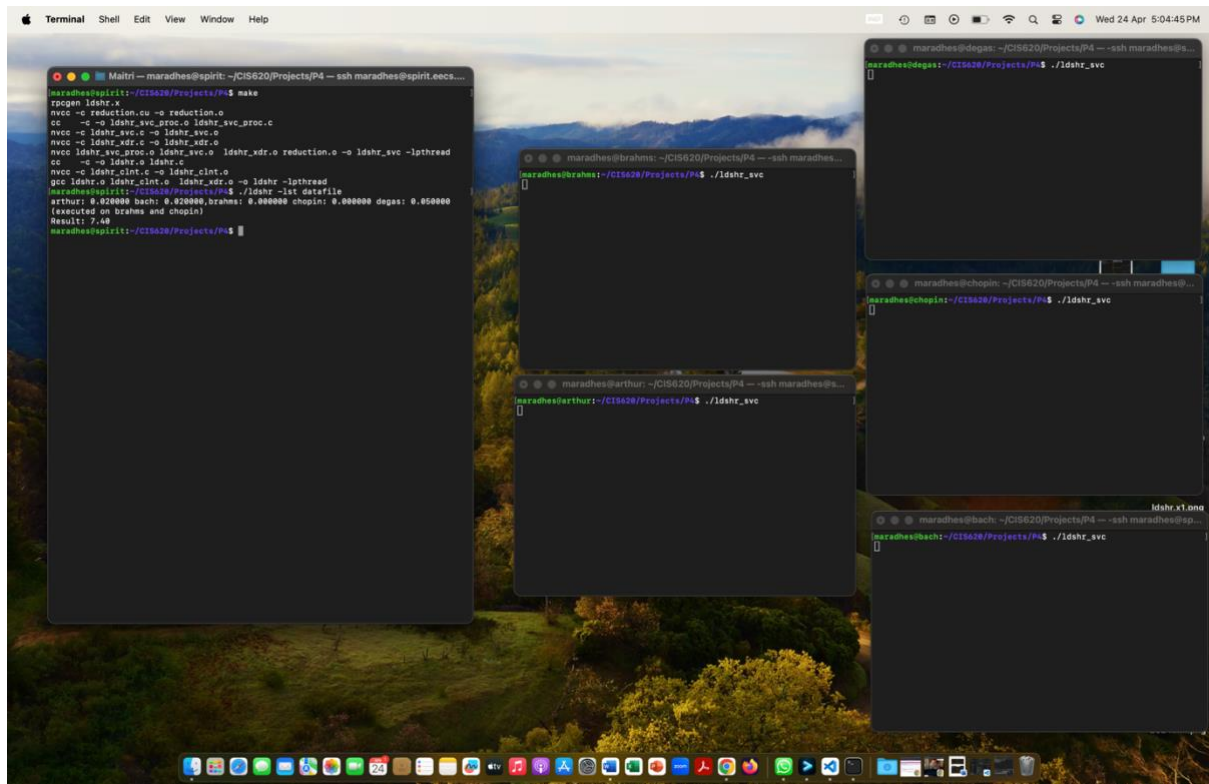
### ./ldshr -gpu 20 5 17 23



- The output on each server window is showing the load averages for each server (the 2 servers with the least loads), and the "**GPU sum**" is being computed in the same two servers which is being picked by the (*'getloadavg()*') which represents the computed result from the **GPU** calculation for each server. Each server is computing its own part of the workload.
- The results from each server (the 2 servers with the least loads) where the **GPU** computations happen are being received by the client (*ldshr.c*) after the *pthread\_join* calls.
- The final summation is aggregating the results from each thread to produce the final output.
- The wrapper function is used from the client side and the function is used to go to the server side to get calculated by the **GPU** (*reduction.cu*).

## CASE 2:

### **./ldshr -lst datafile**



The **-lst** option is provided, it triggers the following:

- Reading a data file and dividing its contents into two linked lists (by alternating between them for each data point).
- Two separate threads are created for processing these lists on the two selected servers.
- Each thread uses the **'sumqroot\_lst\_1Q'** RPC call to process its respective list and compute a sum of quadruple roots.
- After both threads finish, their results are combined to produce a final sum.
- The quadruple root is calculated in map function in **ldshr\_svc\_proc.c**. The reduce function in the same file is used to add the quadruple roots.