Parallel/Distributed Computing (CSEG414/CSE5414)

Assignment #1
이세진 20161619

1. (10 Points) Suppose $T_{serial} = n$ and $T_{parallel} = n/p + log_2(p)$, where times are in microseconds. If we increase p (# of processors) by a factor of k, find a formula for how much we'll need to increase n in order to maintain constant efficiency. How much should we increase n by if we double the number of processes p from 8 to 16? Determine whether this parallel program is (weakly) scalable or not. Hint: Suppose we increase the number of processors, p. If we can find a corresponding rate of increase in the problem size so that the program always has the same efficiency E, then the program is (weakly) scalable.

프로그램이 Weakly scalable 하려면 problem size와 processor의 수를 일정하게 증가하더라도, Efficiency 가 변하지 않아야한다. 따라서

$Efficiency = \frac{Speedup}{p}$

$Speedup = \frac{T_{serial}}{T_{parallel}} = \frac{n}{n/p + log_2(p)}$ 이므로 이를 다시 substitute 하면

$Efficiency = \frac{n}{p(n/p + log_2(p))} = \frac{n}{n + p\,log_2(p))}$ 이다.

n은 m배를 해주고 p 는 k배를 해주면

$Efficiency = \frac{mn}{mn + kp\,log_2(kp))}$ 이며 이는 constant 해야 하므로 즉,

$Efficiency = \frac{n}{n + p\,log_2(p))} = \frac{mn}{mn + kp\,log_2(kp))}$ 이다.

이때 m에 대해서 식을 정리하면, $\frac{1}{m(n+plog_2(p))} = \frac{1}{mn + kp\,log_2(kp))}$

$m(n + plog_2(p)) = mn + kplog_2(kp)$

$m * plog_2(p) = klog_2(kp)$

즉, $m = \frac{klog_2(kp)}{log_2(p)}$ 배 만큼 n을 증가시키면 Efficiency 는 constant 해진다.

두번째로, p가 8에서 16으로 바뀐다면

$m = \frac{2log_2(16)}{log_2(8)} = (2 * 4)/(3) = 8/3$

따라서, n은 $8/3$ 배 만큼 커져야한다. 이 문제에서는 problem size n을 fix 하지 않고 # of processor와 같이 증가시키면서 efficiency가 constant한지 계산하기 때문에 **weakly scalable**하다.

2. (40 Points) Finding prefix sums is a generalization of global sum. Rather than simply finding the sum of n values, X0 + X1 + X2 + ... + Xn-1, the prefix sums are the n partial sums X0, X0 + X1, X0 + X1 + X2, ..., X0 + X1 + ... + Xn-1. MPI provides a collective communication function, MP I_Scan, that can be used to compute prefix sums.

(1) Understand the semantics of MPI_Scan operation and devise at least two parallel prefix sum algorithms (i.e., Explain the algorithms without MPI notation).

MPI_Scan이란 MPI Reduction인 collective communication 함수중 하나이며 프로세스 i의 receive buffer에 프로세스 0에서 프로세스 i까지의 send buffer 데이터들에 대한 reduction 값을 저장한다
MPI_Scan은 collective communication이기 때문에 send, receive 처럼 루프를 사용해 여러번 보낼 필요 없이 함수를 한번만 사용하여 쉽게 구현할 수 있었다.

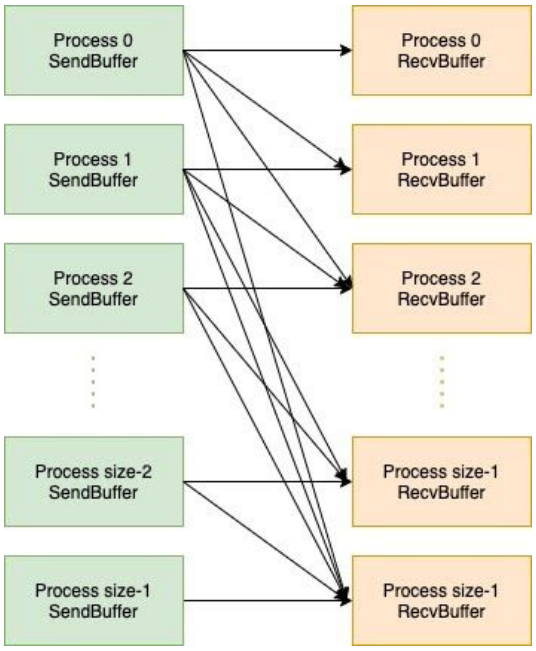
```c
#include <time.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv){
 int rank, size;
 double startTime,endTime;
 int sbuf, rbuf;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 startTime = MPI_Wtime();
 srand(time(NULL) + rank);
 sbuf = rand() % 10; //[-10,10] random integer
 printf("Process %i generated number : %d\n", rank, sbuf);
 MPI_Scan(&sbuf, &rbuf, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
 printf("Process %i's partial sum using MPI_Scan : %d\n", rank, rbuf);
 endTime = MPI_Wtime();
 MPI_Finalize();
 printf("Process %i's execution Time: %lf\n", rank, endTime - startTime);
 return 0;
}
```

(2) Implement this operation using only MPI send and receive (blocking and non blocking) calls. When implementing your solutions, make sure that your implementation is not dependent on the number of processors used. Verify your results by generating n random integers and compare the performance with that of original MPI_Scan as you increase the number of nodes involved and n. Discuss the results.
MPI_Send 와 MPI_Receive를 사용한 방식은 MPI_Scan 과 다르게 point-to-point communication 방식이기 때문에 두 개의 프로세서가 서로와 통신을 하기 위해서 Send, Receive 함수를 한번씩 사용해야 한다. Send 같은 경우 process 0 은 size-1 만큼 모든 프로세스에게 보내고 마지막 프로세서 같은 경우 자기 자신에게만 보내게 된다. 따라서 loop를 사용하여 destination을 자신의 rank에서부터 rank size-1까지 MPI_Send 를 수행한다. 아래의 코드는 blocking send, receive를 사용한 알고리즘이다.

```c
int main(int argc, char **argv){
```

```
int rank, size;
double startTime,endTime;
int sbuf, rbuf, temp;
MPI_Status status;
MPI_Request req;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
startTime = MPI_Wtime();
srand(time(NULL) + rank);
sbuf = rand() % 50;
printf("Process %i generated number : %d\n", rank, sbuf);
for(int dest = rank; dest <= size-1; dest++){ // send to from its rank to n-1
  MPI_Send(&sbuf, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
}
for(int src = rank; src >= 0; src--){
  MPI_Recv(&temp, 1, MPI_INT, src, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  rbuf += temp;
}
printf("Process %i's partial sum using MPI_Scan : %d\n", rank, rbuf);
endTime = MPI_Wtime();
MPI_Finalize();
printf("Process %i's execution Time: %lf\n", rank, endTime - startTime);
return 0;
}
```

이를 그림을 사용하여 표현하면 다음과 같다.



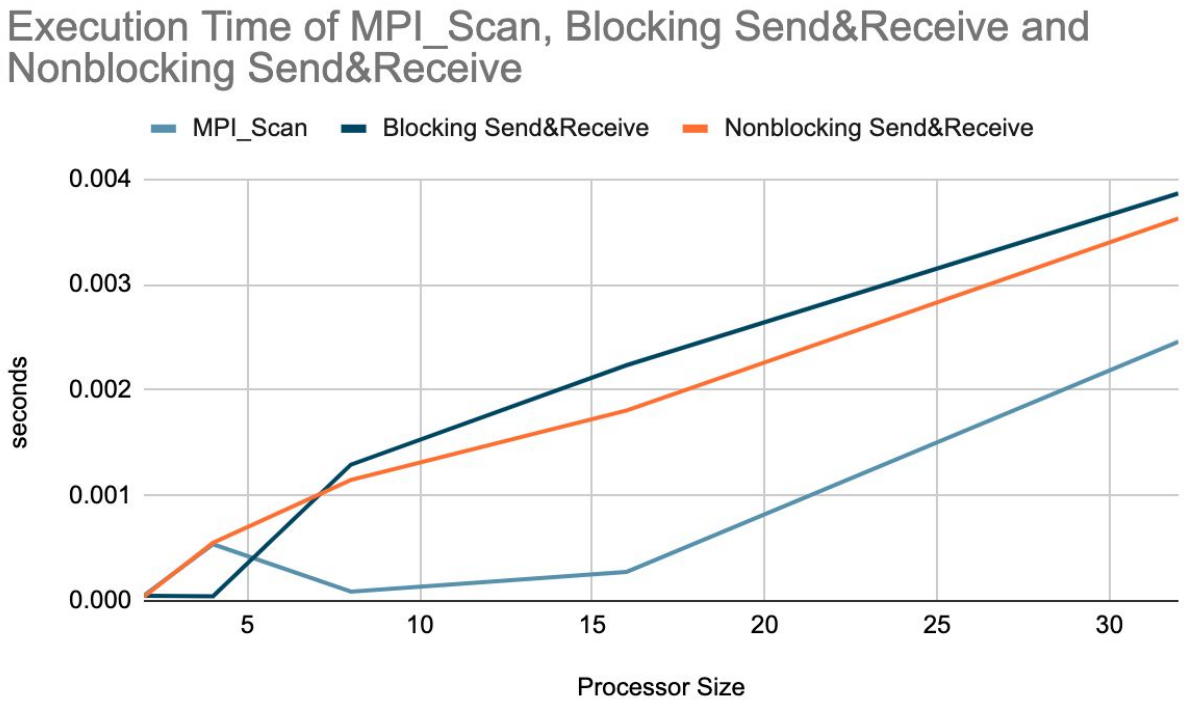
Nonblocking Send, Receive는 동일한 알고리즘을 사용하지만 MPI_Isend, MPI_Irecv 함수를 사용하여 함수가 system call이 되자마자 바로 return을 하여 다른 일을 할 수 있게 한다. synchronization를 맞추기 위해 MPI_Waitall 함수를 사용하였다.

```
int main(int argc, char **argv){
 int rank, size;
 double startTime,endTime;
 int sbuf, rbuf, temp;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Status status[size];
 MPI_Request req[size];
 startTime = MPI_Wtime();
 srand(time(NULL) + rank);
 sbuf = rand() % 50;
 printf("Process %i generated number : %d\n", rank, sbuf);
 for(int dest = rank; dest <= size-1; dest++){ // send to from its rank to n-1
   MPI_Isend(&sbuf, 1, MPI_INT, dest, 0, MPI_COMM_WORLD,&req[rank]);
 }
 for(int src = rank; src >= 0; src--){
   MPI_Irecv(&temp, 1, MPI_INT, src, MPI_ANY_TAG, MPI_COMM_WORLD, &req[src]);
   rbuf += temp;
 }
 MPI_Waitall(rank+1,req,MPI_STATUS_IGNORE);
 printf("Process %i's partial sum using MPI_Scan : %d\n", rank, rbuf);
 endTime = MPI_Wtime();
```

```
    MPI_Finalize();
    printf("Process %i's execution Time: %lf\n", rank, endTime - startTime);
    return 0;
}
```

다음으로, MPI_Scan과 Send,Receive 방식의 성능을 비교해 보았다.

Processor Size	MPI_Scan	Blocking Send&Receive	Nonblocking Send&Receive
2	0.000053	0.000048	0.000041
4	0.000537	0.000044	0.000550
8	0.000088	0.001292	0.001148
16	0.000275	0.002236	0.001806
32	0.002458	0.003865	0.003628



3가지 partial sum 방식의 execution time을 기록하여 그래프로 나타내었다. 세로축은 execution time, 가로축은 processor의 갯 수이다. Partial sum은 프로세서가 많아질수록 problem size도 같이 늘어나기 때문에 execution time이 프로세서가 많아진다고 짧아지지 않았다. 그래프를 보면,

process의 수가 2,4,8,16 일때는 3가지 방식 모두 비슷한 execution time을 가졌다. 하지만, 프로세서의 수가 더 커질수록 MPI_Scan이 Non-blocking, blocking send와 receive 방식보다 더 높은 성능을 가졌다. 이는 MPI_Scan 함수는 reduction 함수로 $log_2(n)$ 번의 송신을 할것으로 예상된다. 한편, MPI_Send, MPI_Recv를 사용한 방법은 프로세서의 수 n번만큼 send를 무조건 보내기 때문에 프로세서가 커질수록 MPI_Scan과 성능 차이가 나타날 것이다.

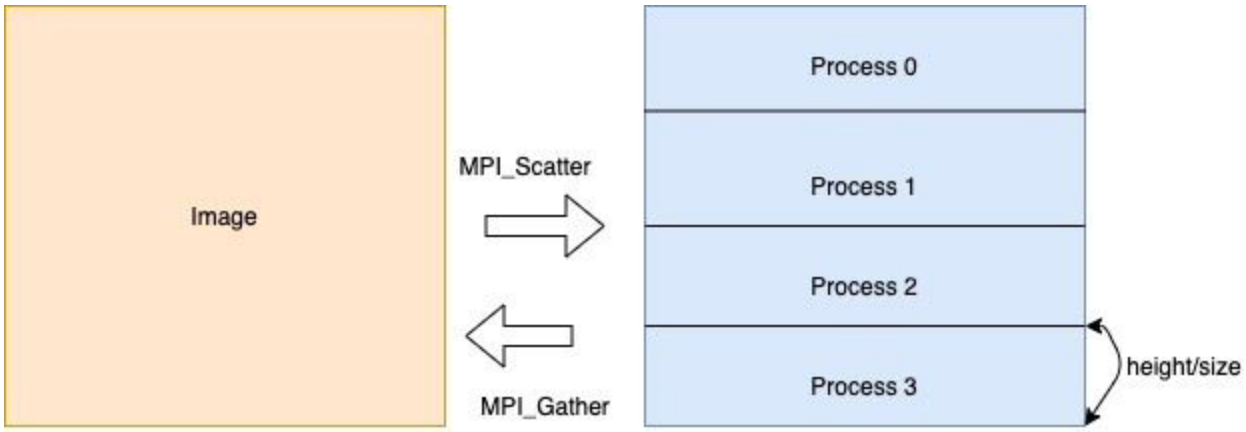
3. (75 Points) It is generally agreed that topics in image processing have the high potential for significant parallelism. In this question, you are to read in a PPM (Portable PixMap) file in P6 format (full color), and write sequential and parallel programs written in C and MPI to

(a) flip an image horizontally (mirroring) and

(b) reduce the image to grayscale by taking the average of the red, green, and blue values for each pixel and

(c) smooth the image by calculating the mean of each pixel's value and its eight neighbors (some algorithms consider only the values from the diagonal neighbors or the horizontal and vertical neighbors).

When implementing your MPI programs, try to use MPI derived data types as much as you can. Compare the performance of sequential and parallel versions of the program and discuss the results over a cluster of workstations. Use different PPM files with various data sizes and discuss the scalability aspects of your code as you increase the number of nodes. When you submit your code, include the short report on the questions above, the sample PPM files used and your programs (sequential and parallel versions). Also include the name of the PPM viewer(Linux version) in a readme file.



병렬화를 하기 위해서 MPI_Type_create_struct(3,len,displ,types,&Pixel) 를 사용하여 pixel의 R,G,B 값을 저장하는 구조체를 만들었다.

Sequential 프로그램과 다르게 Parallel 프로그램은 데이터를 쪼개서 각 프로세서가 image processing을 수행한다. 이때, width는 전체 image width로 고정 시키고 height 만 전체 image height / size로 만들어 배분 해주었다.

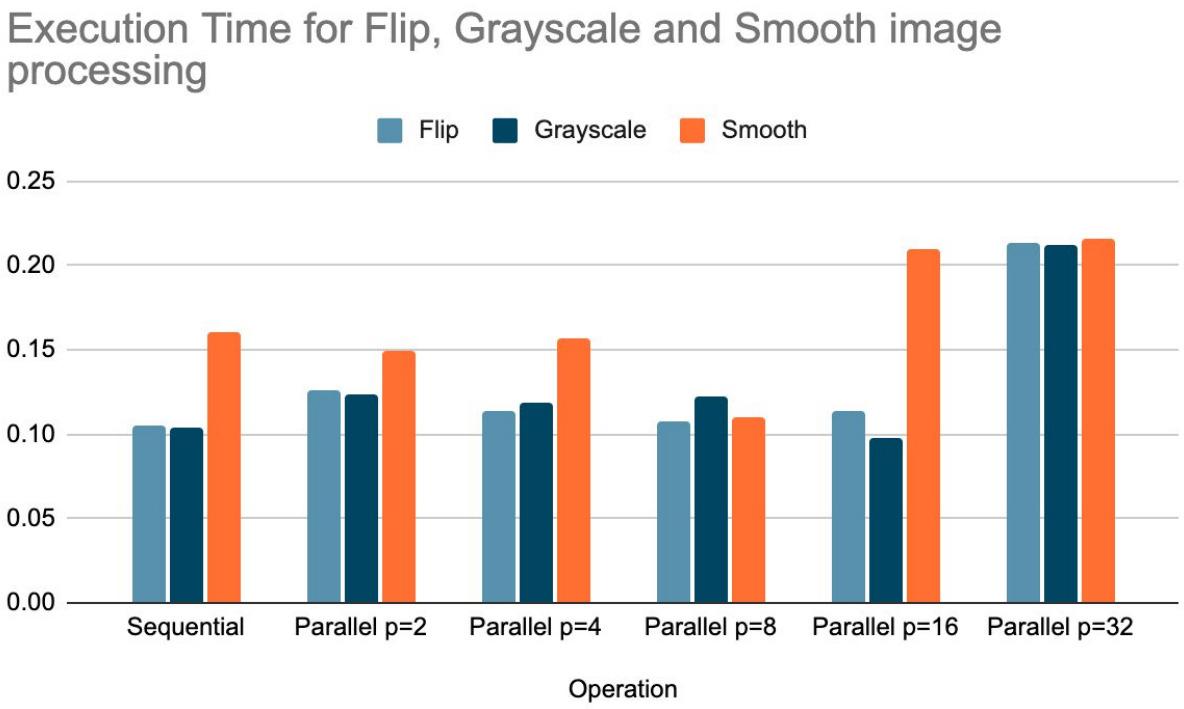
```
int indHeight = h/size;
```

```
dest = (pixel*)malloc(sizeof(pixel)*w*h);
pixel * rbuf = (pixel*)malloc(w * indHeight * sizeof(pixel));
MPI_Scatter(src, w*indHeight, Pixel, rbuf, w*indHeight, Pixel, 0, MPI_COMM_WORLD);

switch(option){
  case 1:
    src = flip(w, indHeight, rbuf);
    break;
  case 2:
    src = grayScale(w, indHeight, rbuf);
    break;
  case 3:
    src = smooth(w, indHeight, rbuf);
    break;
  default:
    printf("Invalid command!\n");
    break;
}
MPI_Gather(src, w*indHeight, Pixel, dest, w*indHeight, Pixel, 0, MPI_COMM_WORLD);
```

Sequential 한 프로그램과 Parallel프로그램의 성능 비교를 하기 위해 Iggy.1024.ppm 파일에 대해 3가지 operation(Flip,grayscale,smooth)의 execution time을 기록하였고 이는 다음과 같다.

	Execution time(seconds)					
Operation	Sequential	Parallel (p=2)	Parallel (p=4)	Parallel (p=8)	Parallel (p=16)	Parallel (p=32)
Flip	0.10561	0.126499	0.114215	0.107592	0.113415	0.212929
Grayscale	0.103685	0.123167	0.118684	0.122507	0.098115	0.212527
Smooth	0.160103	0.14895	0.156379	0.110313	0.210226	0.216455



그래프를 보면, 병렬화를 한 경우 processor의 개 수를 늘릴수록 실행 시간이 조금 더 단축되는 것을 알 수 있다. 하지만 processor의 수가 어느 시점보다 더 커지면(약 16부터) 처리하는 데이터 수에 비해 병렬화를 하는 overhead가 더 커져 오히려 실행 시간이 더 오래 걸리는, 즉 성능이 떨어지는 것을 확인할 수 있었다. 따라서, process 수가 증가해도 efficiency 가 constant하지 않은걸 알 수 있다. 따라서, strong scalable 하지 않다는걸 알 수 있었다.

Operation	Problem Size (Image Size)		
	133*133 (53KB)	800*600(1.4MB)	1024*1024(3.1MB)
Flip	0.005923	0.092713	0.107592
Grayscale	0.006026	0.092325	0.122507
Smooth	0.007444	0.097680	0.110313

마지막으로, processor 수를 8로 고정시키고 problem size(즉, 이미지의 사이즈) 를 바꾸어 실험해 보았다. 이처럼 1.4 MB 이미지와 3.1MB 이미지는 problem size가 약 2배 차이 나지만 execution time은 거의 동일한걸 알 수 있었다.

4. (75 Points) In this assignment, you are required to implement a calculating server and its corresponding client program using SUN RPC. The calculating server will provide

service procedures such as addition(+), subtraction(-), multiplication (*), division(/), and x power y(xy) of any two integer values. In the client side, you can implement a command-line calculator that accepts an expression (consisting of only integer values and five arithmetic operators (+, -, *, /, **)) and prints out the corresponding answer.

작성한 프로그램은 server program에서는 연산자와 두 integer 정수 값을 받고 결과 값을 계산한다.
Client program에서는 string으로 받은 식을 통해 숫자와 연산자를 구분한 뒤 사칙 연산 순서를 계산하고 서버에게 request를 보내서 최종 결과값을 출력한다.

클라이언트에서 서버에게 request를 보낼 때는 하나의 parameter로 송신해야 하기 때문에 inputs이라는 구조체를 만들어서 contract에 작성하였다. 아래는 작성한 contract인 calc.x 이다.

```
struct inputs{
    int n1;
    int n2;
    char operator;
};
program CALCULATOR_PROG{
    version CALCULATOR_VERS{
            int server_calculator(inputs)=1;
    }=1;
}=0x31111111;
```

rpcgen -a -C rand.x 명령어를 통해

```
Makefile.calc  calc.x       calc_client.c  calc_clnt.c
calc.h       calc_server.c  calc_svc.c     calc_xdr.c
```

calc_client.c, calc_server.c 를 포함한 총 7개의 파일을 생성하여 client 프로그램을 calc_client.c, server 프로그램은 calc_server.c에 작성하였다.

아래는 calc_server.c 코드이다. 서버 프로그램은 server_calculator 라는 한가지의 method을 제공하며 이는 두 개의 integer과 한 가지의 operator(+, - , * , / , ^) 를 받아 결과값을 return 한다.

```
#include "calc.h"
int *
server_calculator_1_svc(inputs *argp, struct svc_req *rqstp)
{
    static int  result;
    result=0;
    if(argp->n2==0 && argp->operator=='/'){
        printf("ERROR: cannot divide by 0; we will calculate this as 0\n");
        return &result;
    }
    switch(argp->operator){
            case '+': result= argp->n1+argp->n2;break;
            case '-': result= argp->n1-argp->n2;break;
```

```
19             case '*': result= argp->n1*argp->n2;break;
20             case '/': result= argp->n1/argp->n2;break;
21             case '^': result= 1;
22                       for(int i =0 ; i < argp->n2; i++){
23                           result *= argp->n1;
24                       }
25                       break;
26     }
27
28     printf("Server calculated .. %d %c %d = %d\n",argp->n1,argp->operator,
argp->n2,result);
29     return &result;
30 }
```

한편, clients는 사용자에게 받은 데이터, 즉 string에서 숫자와 연산자를 구분하고 연산처리 순서를 계산한다.
seperate() 함수에서 유저의 input으로부터 숫자와 연산자를 구분하고 아래와 같은 구조체로 바꾸어 저장을 한다. 문자열을 읽으면서 숫자면 isNumber = 1 과 n에 숫자를 저장하고 반대로 연산자이면 isNumber = 0, op 에 연산자를 저장한다.

```
 6 typedef struct expr{
 7     int n;
 8     char op;
 9     int isNumber;
10 }expr;
```

```
33 int seperate(char * expression){
34     int previousNum = 0;
35     int previousIndex = 0;
36     int isNegative = 1;
37     int precendence;
38     int num = 0;
39     int consec = 0;
40     expr e;
41     for(int i = 0; i < strlen(expression); i++){
42         char c = expression[i];
43         precendence = getPrecedence(c);
44         if(precendence == -1 ){
45             return 0;
46         }
47         if(!precendence){ // is a number
48             previousNum = 1;
```

```
49              num*=10;
50              num+=atoi(&c);
51              consec = 0;
52          }else{ // operator
53              if(previousNum){ // store number
54                  e.isNumber = 1;
55                  e.n = num * isNegative;
56                  isNegative = 1;
57                  ex[total++] = e;
58                  num = 0;
59              }
60              if(consec == 2 ){ // more than 3 operators in consecutive order
61                  printf("Error! Format of operation\n");
62                  return 0;
63              }
64
65              if((i==0 || getPrecedence(expression[i-1])) && expression[i] == '-') {
66                  isNegative = -1;
67              }else{
68                  if(i!=strlen(expression)-1 && expression[i+1] == '*' &&
expression[i]=='*'){
69                      e.op='^';
70                      e.isNumber = 0;
71                      ex[total++]=e;
72                      i+=1;
73                  }else{
74                      e.op=c;
75                      e.isNumber = 0;
76                      ex[total++]=e;
77                  }
78              }
79              previousNum = 0;
80              consec += 1;
81          }
82      }
83      if(precendence){
84          printf("Error! expression cannot end with an operator\n");
85          return 0;
86      }
87      e.isNumber = 1;
88      e.n = num * isNegative;
89      ex[total++]=e;
```

```
 90     if(total==1){
 91         printf("Error! No operator given.\n");
 92         return 0;
 93     }
 94     return 1;
 95 }
```

이처럼 expr 구조체로 변환한뒤에 배열에 저장한뒤엔 operator를 저장하는 stack 하나와 숫자를 저장하는 stack 하나를 사용하여 사칙연산을 한다. 이는 calculator_prog_1() 함수에서 수행된다. 구조체 배열을 순서대로 읽어주며 숫자면 숫자 stack에 push를 해준다. 반대로 연산자면 연산자 stack의 top 과 비교하여 precedence 가 작거나 같다면 숫자 stack에서 숫자 2개를 pop() 하고 연산자를 pop() 하여 연산을 한다. 이때, 사칙 연산을 하는 함수는 server에 있으므로 serer_calculator_1_arg 에 숫자 2개와 연산자를 넣어주고 server_calculator_1(&server_calculator_1_arg,clnt) 함수를 사용했다. 그 후, 얻은 결과 값을 다시 숫자 스택에 넣어준다. 구조체 배열을 다 읽고 나서, 연산자 stack이 empty 할때까지 같은 과정을 반복한다. 연산자 stack이 empty 가 되면, 숫자 stack 의 top 에 있는 숫자가 바로 최종 결과값이다.

```
 97 void calculator_prog_1(char *host)
 98 {
 99     CLIENT *clnt;
100     int  *result_1;
101     char op;
102     int n1,n2;
103     inputs  server_calculator_1_arg;
104 
105 #ifndef DEBUG
106     clnt = clnt_create (host, CALCULATOR_PROG, CALCULATOR_VERS, "udp");
107     if (clnt == NULL) {
108         clnt_pcreateerror (host);
109         exit (1);
110     }
111 #endif  /* DEBUG */
112     for(int i = 0 ; i < total; i++){
113         if(ex[i].isNumber){
114             numberStack[++numHead] = ex[i].n;
115         }else{
116             char c = ex[i].op;
117             while(opHead >= 0 && getPrecedence(operatorStack[opHead]) >=
getPrecedence(c)){
118                 if(numHead <= 0 ) {
119                     printf("Error! operation format!\n");
```

```
120                 }
121                 n2 = numberStack[numHead--];
122                 n1 = numberStack[numHead--];
123                 op = operatorStack[opHead--];
124                 if(op=='/' && n2==0){
125                     printf("Error! Cannot divide by zero\n");
126                     return;
127                 }
128                 server_calculator_1_arg.n2 = n2;
129                 server_calculator_1_arg.n1 = n1;
130                 server_calculator_1_arg.operator = op;
131                 result_1 = server_calculator_1(&server_calculator_1_arg,clnt);
132                 if(result_1 == (int*)NULL){
133                     clnt_perror(clnt,"call failed");
134                 }
135 
136                 numberStack[++numHead] = *result_1;
137             }
138             operatorStack[++opHead]= ex[i].op;
139         }
140         }
141         while(opHead>=0){
142             if(numHead<=0){
143                 printf("Error! in operation format\n");
144                 return;
145             }
146             server_calculator_1_arg.n2 = numberStack[numHead--];
147             server_calculator_1_arg.n1 = numberStack[numHead--];
148             server_calculator_1_arg.operator = operatorStack[opHead--];
149             if(server_calculator_1_arg.n2 == 0 && server_calculator_1_arg.operator
==='/'){
150                 printf("Error! Cannot divide by zero\n");
151                 return;
152             }
153             result_1 = server_calculator_1(&server_calculator_1_arg,clnt);
154             numberStack[++numHead] = *result_1;
155         }
156 
157         printf("result is %d\n", numberStack[numHead]);
158 #ifndef DEBUG
159     clnt_destroy (clnt);
160 #endif   /* DEBUG */
```

```
161 }
```