# 8 - WEEK REPORT

**TITLE:-** Concurrent Data structures and Multi-threading
**NAME:-** Siddharth Shukla (ENGS686)
**NAME OF THE GUIDE:-** Dr. Sathya Peri
**INSTITUTION OF GUIDE:-** IIT Hyderabad

## ABSTRACT:-

In this final report following problems that were implemented and analysed during the internship have been discussed.

- Producer-Consumer problem using semaphores.
- Solving dense system of linear algebraic equations on a multi-processor system using:
  - ***-Semiring based algorithm**.
  - **Successive Gauss Elimination algorithm**.
- Multiplication of 2 matrices using static as well as dynamic allocation of threads.
- Efficient chain matrices multiplication through dynamic programming using multithreading.

## INTRODUCTION:-

Recently, shared memory multiprocessors have been used to implement a wide range of high performance applications. The use of multithreading to program such applications is becoming popular, and POSIX threads or Pthreads are now a standard supported on most platforms.

Pthreads may be implemented either at the kernel level, or as a user-level threads library. Kernel-level implementations require a system call to execute most thread operations, and the threads are scheduled by the kernel scheduler. This approach provides a single, uniform thread model with access to system-wide resources, at the cost of making thread operations fairly expensive. In contrast, most operations on user-level threads, including creation, synchronization and context switching, can be implemented in user space without kernel intervention, making them significantly cheaper than corresponding operations on kernel threads. Thus parallel programs can be written with a large number of lightweight, user-level Pthreads, leaving the scheduling and load balancing to the threads implementation, and resulting in simple, well structured, and architecture-independent code. A user-level implementation also provides the flexibility to choose a scheduler that best suits the application, independent of the kernel scheduler.

## PROBLEMS:

- **Producer-Consumer problem:** This problem is one of the small collection of standard, well-known problems in concurrent programming: a finite-size buffer and two classes of threads, producers and consumers, put items into the buffer (producers) and take items out of the buffer (consumers). To implement the bounded buffer producer-consumer problem using semaphores. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

The following C code was successfully implemented as the solution of fore-mentioned using Pthreads on g++ compiler on a dual core system with Linux OS.

```
#include <stdio.h>
#include<stdlib.h>
#include <sys/types.h>
#include<pthread.h>
#include <semaphore.h>
#include <time.h>

sem_t mut;
sem_t empty;
sem_t full;
int buffer[200];
pthread_t ptd[20],ctd[20];
int cap,cntp,cntc,up,uc,bi=0,bo=0;

void *producer(void *arg){
int i,item,ID=pthread_self();
for(i=1;i<=cntp;i++){
item=1+(rand() % 500);
sem_wait(&empty);
sem_wait(&mut);
bi=(bi+1)%cap;

buffer[bi]=item;
time_t dur=time(NULL);
printf("%dth item: %d produced by thread %d at %s into buffer location %d\n",i,item,ID,asctime(gmtime(&dur)),bi);
sem_post(&mut);
sem_post(&full);
}}

void *consumer(void *arg){
int i,item,ID=pthread_self();
for(i=1;i<=cntc;i++){
sem_wait(&full);
sem_wait(&mut);
item=buffer[bo+1];
bo=(bo+1)%cap;
time_t dur=time(NULL);
printf("%dth item: %d consumed by thread %d at %s from buffer location
%d\n",i,buffer[bo],ID,asctime(gmtime(&dur)),bo);
sem_post(&mut);
sem_post(&empty);
}}

int main(){
int i,np,nc;
printf("Enter capacity,np,nc,cntp,cntc,up,uc: ");
scanf("%d %d %d %d %d %d %d",&cap,&np,&nc,&cntp,&cntc,&up,&uc);
sem_init(&mut,0,1);sem_init(&empty,0,cap);sem_init(&full,0,0);
for(i=1;i<=np;i++)
pthread_create(&ptd[i],NULL,producer,NULL);
for(i=1;i<=nc;i++)
pthread_create(&ctd[i],NULL,consumer,NULL);
for(i=1;i<=np;i++)
pthread_join(ptd[i],NULL);
for(i=1;i<=nc;i++)
pthread_join(ctd[i],NULL);
exit(0);}
```

The order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock.

- **Solving dense system of linear algebraic equations on a multi-processor system using *-semiring based algorithm:** We have to determine the unknown solution column vector X, in the linear algebraic equation AX = B (where A is a real non-singular matrix of order N and B is a known right hand side vector). The required research paper**[1]** can be found under references section.

  ∗-semirings are algebraic structures that provide a unified approach to solve several problem classes in computer science and operations research. Matrix computations over ∗-semirings are interesting because of their potential applications to linear algebra.The concept of **eliminant** is introduced to give closed form expressions for describing solutions of linear equations over *-semirings. This method computes solution vector X in AX=B without explicitly finding inverse of A (but computes it implicitly).

The successfully implemented sequential as well as parallelized C++ code for this problem is given below, with results following them.

Sequential code:

```
#include <bits/stdc++.h>
#include <pthread.h>
#include <time.h>
#include <semaphore.h>
#define n 1000
#define ll int
using namespace std;
float X[2*n][n+1];

void soln(){
```

```
for(ll t=0;t<n;t++){
float temp=(float)(1/(1-X[t][t]));
for(ll i=t+1;i<=n+t;i++)
for(ll j=t+1;j<=n;j++)
X[i][j]+=(X[i][t] * X[t][j] * temp);
}}

int main(){
clock_t tStart = clock();
ll a[n][n],b[n][1];
int k=0,j=0,i;
for(ll i=0;i<n*n;i++){
a[j][k]=rand()%40;
k++;
if(k==n){k=0;j++;}}
for(i=0;i<n;i++){
b[i][0]=rand()%50;}
for(i=0;i<2*n;i++){
for(j=0;j<=n;j++){
if(j<n && i<n){
if(i==j){
X[i][j]=1-a[i][j];}
else X[i][j]=(-a[i][j]);}
else if(j==n && i<n)X[i][j]=b[i][0];
else{
if((i-j)==n && i>=n)X[i][j]=1;
else X[i][j]=0;
}}}
soln();
cout<<"\nValues of x are as follows: \n";
for(i=0;i<n;i++)
cout<<"x"<<i+1<<"= "<<X[i+n][n]<<"\n";
printf("\nTime taken: %.6fs\n",(clock() - tStart)/CLOCKS_PER_SEC);
exit(0);}
```

Parallelized code (for 100 unknowns in each of the 100 algebraic equations & 8 threads):

```
#include<bits/stdc++.h>
#include <pthread.h>
#include <semaphore.h>
using namespace std;
#define N 100
#define M 8
int a[N][N],b[N][M];
double x[2*N][N+1];
pthread_t mat[N];
int k=0,n,y;
void *matrix(void *arg)
{  if(y < (n-1)){
       double temp=1/(1-x[k][k]);
       for(int i=k+1;i<=N+k;i++)
       { for(int j=k+1;j<=N;j++)
         {x[i][j]=(double)(x[i][j]+(x[i][k]*x[k][j]*temp));
            } }
       ++k;}
        else
        {  for(int l=k;l<=(N-1);l++){
            double temp=1/(1-x[l][l]);
            for(int i=l+1;i<=N+l;i++)
            { for(int j=l+1;j<=N;j++)
               {x[i][j]=(double)(x[i][j]+(x[i][l]*x[l][j]*temp)); }
              }}}
}

int main()
{  clock_t tStart = clock();
    int j=0,t=0,n;
    cout<<"Enter number of threads :";
    cin>>n;
  for(int i=0;i<10000;i++)
      {   if(t==99)
             { a[j][t]=rand()%40;
                t=0;j++;}
            else{
               a[j][t]=rand()%40;
            t++;}
        }
   for(int i=0;i<100;i++)
     b[i][M]=rand()%40;
    for(int i=0;i<N;i++)
   {for(int j=0;j<N;j++)
     { if(i==j)
       x[i][j]=(double)1-a[i][j];
        else
        x[i][j]=(double)-a[i][j];
       } }
      for(int i=0;i<N;i++)
        x[i][N]=(double)b[i][0];
      for(int i=N;i<=(2*N-1);i++)
      { for(int j=0;j<=N;j++)
        {
          if(i-N==j)
            x[i][j]=(double)1;
           else
            x[i][j]=(double)0;
        }}
  void *exit_status;
      for (y=0;y<n;y++){
        pthread_create(&mat[y],NULL,matrix,&y);
     }
     for(int i=0;i<n;i++)
      pthread_join(mat[i],&exit_status);
      for(int i=N;i<=(2*N-1);i++)
          cout<<"x["<<i-N<<"]="<<x[i][N]<<endl;
printf("\nTime taken: %.6fs\n"(clock() - tStart)/CLOCKS_PER_SEC);
  return 0;
}
```

For matrix A (in AX=B) with order 100, the running time for the sequential code was 3.39 seconds while that for the parallelized code (running on 8 threads) was 0.028836 seconds. Further, the programs were tested for matrix A of order 1000, to which the parallelized program successfully executed in 7.11 seconds and the sequential program resulted in abnormal termination.

- **Solving dense system of linear algebraic equations on a multi-processor system using Successive Gauss Elimination ( SGE ):** We have to determine the unknown solution column vector X, in the linear algebraic equation AX = B (where A is a real non-singular matrix of order N and B is a known right hand side vector). The required research paper**[2]** can be found under references section.

  Being a variant of Gaussian Elimination (GE), the dependencies of all the unknowns are reduced to half at every stage and finally to zero in log2 N stages (N linear independent equations at Stage 1 are replaced by two sets of N/2 linear independent equations at Stage 2, and further) which is accomplished by using the concept of forward (left to right) and backward (right to left) eliminations. Unlike the conventional GE algorithm, the SGE algorithm does not have a separate back substitution phase, which requires O(N) steps using O(N) processors or O (log 2 N) steps using O (N^3) processors, for solving a system of linear algebraic equations. It replaces the back substitution phase by only one step division (hence constant time) and possesses numerical stability through partial pivoting.
  The problem was implemented recursively as well as iteratively in a sequential as well as parallelized manner.

  **Recursive solution:**
- Sequential code

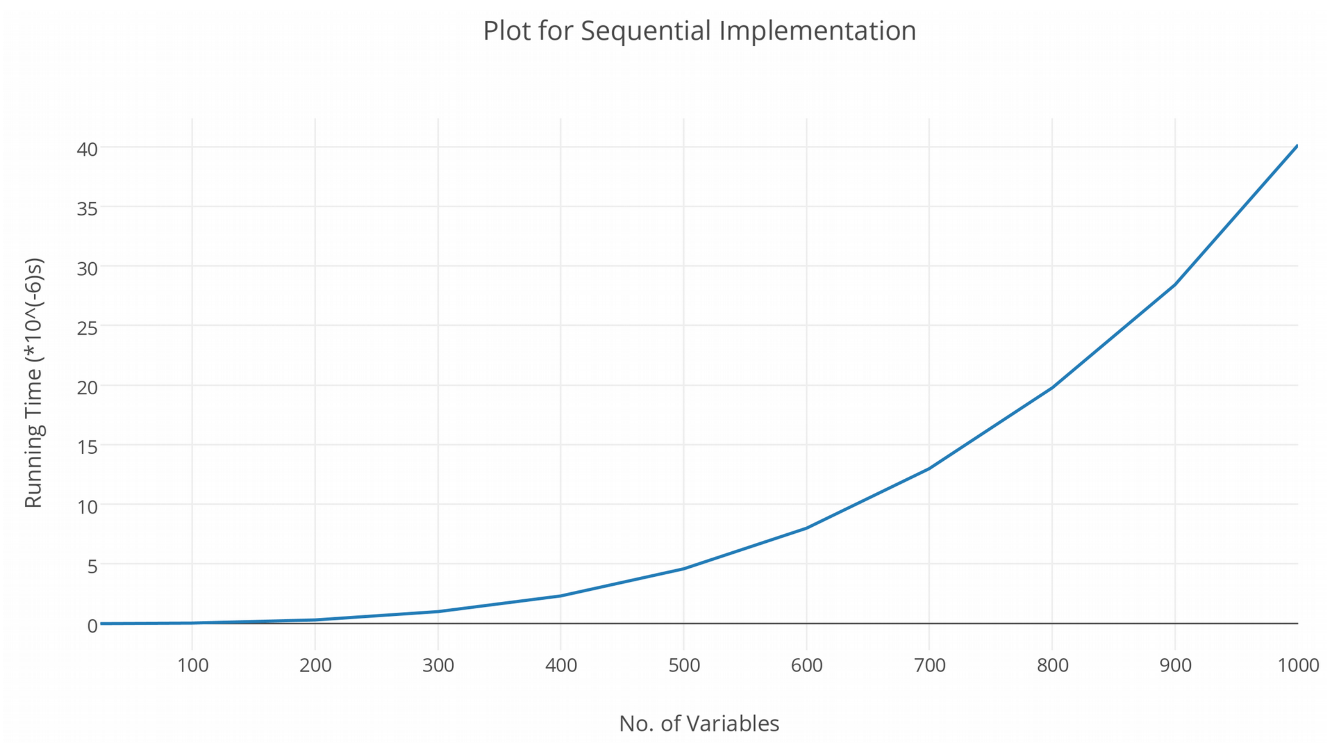
```
#include<bits/stdc++.h>
#include <pthread.h>
#include <time.h>
#include <semaphore.h>
#include<stdlib.h>
#include <sys/types.h>
#define ll long long int
#define pb push_back
using namespace std;
vector<float> ans;

void fwdelim(vector<vector<float> > &c,int n,int m){   //Forward Elimination
int q,i,j,k;
for(k=0;k<(n/2);k++){
float max=c[k][k];
for(i=k;i<n;i++)
if(c[i][k]>=max){max=c[i][k];q=i;}
for(i=0;i<m;i++)
{float g=c[q][i];c[q][i]=c[k][i];c[k][i]=g;}
for(q=k+1;q<n;q++)
c[q][k]=(c[q][k]/c[k][k]);
for(j=k+1;j<m;j++)
for(i=k+1;i<n;i++)
c[i][j]-=((c[i][k])*(c[k][j]));
}}

void bwdelim(vector<vector<float> > &b,int n,int m){   //Backward Elimination
int q,i,j,k;
for(k=m-2;k>=(n/2);k--){
float max=b[k][k];
for(i=k;i>=0;i--)
if(b[i][k]>=max){max=b[i][k];q=i;}
for(i=0;i<m;i++)
{float g=b[q][i];b[q][i]=b[k][i];b[k][i]=g;}
for(q=k-1;q>=0;q--)
b[q][k]=(b[q][k]/b[k][k]);
for(j=k-1;j>=0;j--)
for(i=k-1;i>=0;i--)
b[i][j]-=((b[i][k])*(b[k][j]));

for(i=k-1;i>=0;i--)
b[i][m-1]-=((b[i][k])*(b[k][m-1]));
}}

void SGE(vector<vector<float> > &a,int n,int m){
if(n>1){
vector<vector<float> >a1,a2;
for(int i=0;i<n;i++){vector<float> v;
for(int j=0;j<m;j++){
v.pb(a[i][j]);}
a1.pb(v);a2.pb(v);}
fwdelim(a1,n,m);
bwdelim(a2,n,m);
vector<vector<float> >a3,a4;

for(int i=0;i<n/2;i++){vector<float> v;
for(int j=0;j<n/2;j++)
{v.pb(a2[i][j]);}v.pb(a2[i][m-1]);
a4.pb(v);}

for(int i=n/2;i<n;i++){vector<float> v;
for(int j=n/2;j<n;j++){
v.pb(a1[i][j]);}v.pb(a1[i][m-1]);
a3.pb(v);}

SGE(a3,n/2,n/2+1);
SGE(a4,n/2,n/2+1);
}
else ans.pb(a[0][1]/a[0][0]);
}

int main(){
clock_t tStart = clock();
float q;
int i,j,r=1000,c=1001;
vector<vector<float> >a;
for(i=0;i<r;i++){
vector<float> v;
for(j=0;j<c;j++){
v.pb(rand()%25);
}
a.pb(v);
}
SGE(a,r,c);
for(i=0;i<ans.size();i++)
{printf("x[%d] = %f\n",i+1,ans[i]);}
printf("\nTime taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
exit(0);}
```

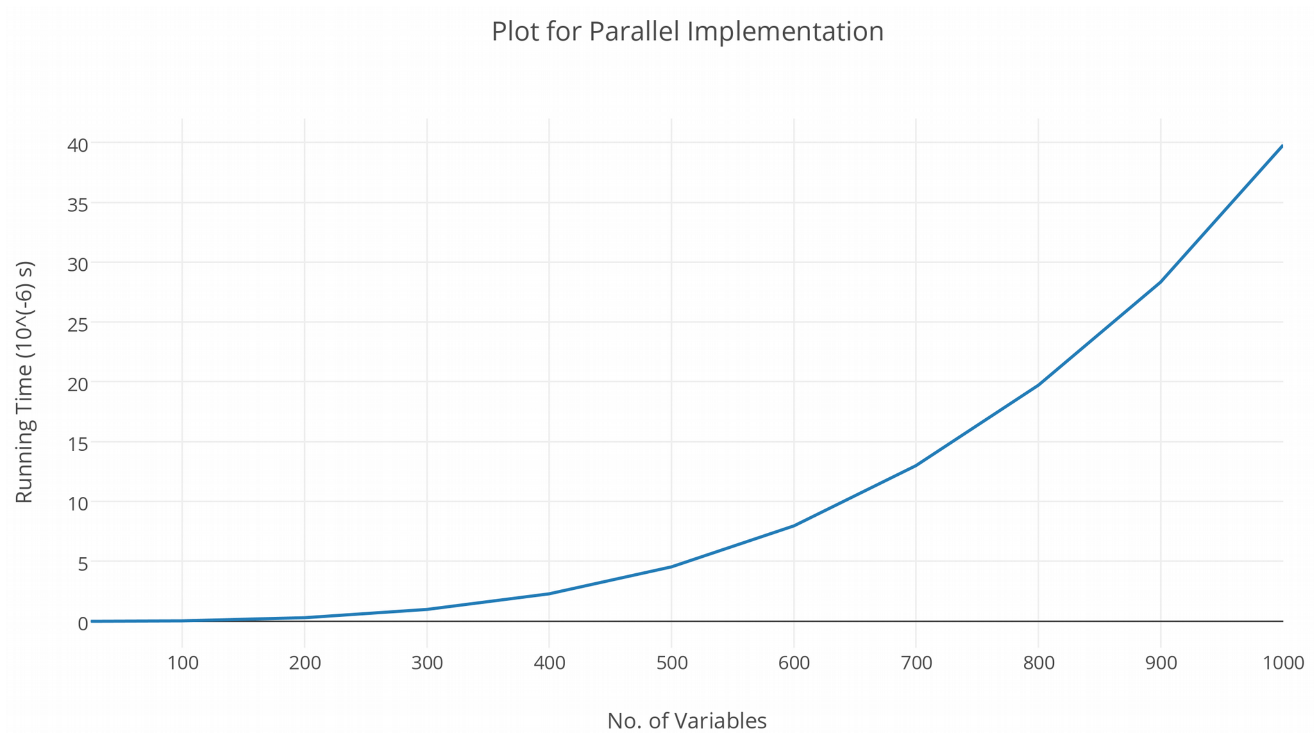
- Parallelized code (using 2 threads)

```
#include<bits/stdc++.h>
#include <pthread.h>
#include <time.h>
#include <semaphore.h>
#include<stdlib.h>
#include <sys/types.h>
#define ll long long int
#define pb push_back
using namespace std;

sem_t mut;
pthread_t ptd[10];
int i,j,r=1000,c=1001;
vector<float> ans;
vector<vector<float> >a;

void fwdelim(vector<vector<float> > &c,int n,int m){ // Forward Elimination
int q,i,j,k;
for(k=0;k<(n/2);k++){
float max=c[k][k];
for(i=k;i<n;i++)
if(c[i][k]>=max){max=c[i][k];q=i;}
for(i=0;i<m;i++)
{float g=c[q][i];c[q][i]=c[k][i];c[k][i]=g;}
for(q=k+1;q<n;q++)
c[q][k]=(c[q][k]/c[k][k]);
for(j=k+1;j<m;j++)
for(i=k+1;i<n;i++)
c[i][j]-=((c[i][k])*(c[k][j]));
}
}

void bwdelim(vector<vector<float> > &b,int n,int m){// Backward Elimination
int q,i,j,k;
for(k=m-2;k>=(n/2);k--){
float max=b[k][k];
for(i=k;i>=0;i--)
if(b[i][k]>=max){max=b[i][k];q=i;}
for(i=0;i<m;i++)
{float g=b[q][i];b[q][i]=b[k][i];b[k][i]=g;}
for(q=k-1;q>=0;q--)
b[q][k]=(b[q][k]/b[k][k]);
for(j=k-1;j>=0;j--)
for(i=k-1;i>=0;i--)
b[i][j]-=((b[i][k])*(b[k][j]));

for(i=k-1;i>=0;i--)
b[i][m-1]-=((b[i][k])*(b[k][m-1]));
}
}

void SGE(vector<vector<float> > &a,int n,int m){
if(n>1){
vector<vector<float> >a1,a2;
for(int i=0;i<n;i++){vector<float> v;
for(int j=0;j<m;j++){
```

```
v.pb(a[i][j]);}
a1.pb(v);a2.pb(v);}
fwdelim(a1,n,m);
bwdelim(a2,n,m);
vector<vector<float> >a3,a4;

for(int i=0;i<n/2;i++){vector<float> v;
for(int j=0;j<n/2;j++)
{v.pb(a2[i][j]);}v.pb(a2[i][m-1]);
a4.pb(v);}

for(int i=n/2;i<n;i++){vector<float> v;
for(int j=n/2;j<n;j++){
v.pb(a1[i][j]);}v.pb(a1[i][m-1]);
a3.pb(v);}

SGE(a3,n/2,n/2+1);
SGE(a4,n/2,n/2+1);
}
else ans.pb(a[0][1]/a[0][0]);
}

void *soln(void *arg){
SGE(a,r,c);}

int main(){
clock_t tStart = clock();
float q;
for(i=0;i<r;i++){
vector<float> v;
for(j=0;j<c;j++){
v.pb(rand()%25);
}
a.pb(v);
}
sem_init(&mut,0,1);
pthread_create(&ptd[0],NULL,soln,NULL);
pthread_join(ptd[0],NULL);
for(i=0;i<ans.size();i++)
{printf("x[%d] = %f\n",i+1,ans[i]);}
printf("\nTime taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
exit(0);}
```

In the recursive solution, the performance gain was quite less (for a matrix of order 1000, the sequential program executed in 40.186076 seconds and the parallelized solution executed in 39.814362 seconds). The performance similarity can be realised using the following line plots drawn using experimental testing of the programs.





Due to the inefficiency of the recursive solution, an iterative solution to the above method was attempted.

**Iterative solution:**

- Sequential code

```
#include<bits/stdc++.h>
```

```
#include <time.h>
#include<stdlib.h>
#include <sys/types.h>
#define ll long long int
#define pb push_back
using namespace std;

int main(){
clock_t tStart = clock();
ll i,j,k,q,r=200,c=201;
vector<vector<float> >A,B,C;
for(i=0;i<r;i++){
vector<float> v;
for(j=0;j<c;j++){
v.pb(rand()%25);
}
A.pb(v);
}
ll l=0,n=r,m=n,t;

while(m>1){
t=0;//for(i=0;i<n;i++){B[i].clear();C[i].clear();}
for(i=0;i<r;i++){
vector<float> v;
for(j=0;j<c;j++){
v.pb(A[i][j]);}B.pb(v);C.pb(v);}

for(t=0;t<(n/m);t++){
for(k=0+t*m;k<((m/2)+t*m);k++){
float max=B[k][k];
for(i=k;i<m+t*m;i++)
if(B[i][k]>=max){max=B[i][k];q=i;}
for(i=0+t*m;i<m+t*m;i++)
{float g=B[q][i];B[q][i]=B[k][i];B[k][i]=g;}
for(q=k+1;q<m+t*m;q++)
B[q][k]=(B[q][k]/B[k][k]);
for(j=k+1;j<m+t*m;j++)
for(i=k+1;i<m+t*m;i++)
B[i][j]-=((B[i][k])*(B[k][j]));
for(i=k+1;i<m+t*m;i++)
B[i][c-1]-=((B[i][k])*(B[k][c-1]));
}//for k

for(i=0+t*m;i<((m/2)+t*m);i++)
for(j=0+t*m;j<((m/2)+t*m);j++)
A[i][j]=B[i][j];
for(i=0+t*m;i<((m/2)+t*m);i++)
A[i][c-1]=B[i][c-1];
}

for(t=0;t<(n/m);t++){
for(k=c-2-t*m;k>=(c-2)-(m/2)-t*m+1;k--){
float max=C[k][k];
for(i=k;i>=t*m;i--)
if(C[i][k]>=max){max=C[i][k];q=i;}
for(i=(c-2)-t*m;i>=(c-2)-(t)*m-m+1;i--)
{float g=C[q+1][i];C[q+1][i]=C[k][i];C[k][i]=g;}
for(q=k-1;q>=(r-1)-t*m-m+1;q--)
C[q][k]=(C[q][k]/C[k][k]);
for(j=k-1;j>=(c-2)-(t)*m-m+1;j--)
for(i=k-1;i>=(r-1)-t*m-m+1;i--)
C[i][j]-=((C[i][k])*(C[k][j]));
for(i=k-1;i>=(r-1)-t*m-m+1;i--)
C[i][c-1]-=((C[i][k])*(C[k][c-1]));
}
for(i=c-2-t*m;i>=(c-2)-(m/2)-t*m+1;i--)
for(j=c-2-t*m;j>=(c-2)-(m/2)-t*m+1;j--)
A[i][j]=C[i][j];
for(i=c-2-t*m;i>=(c-2)-(m/2)-t*m+1;i--)
A[i][j]=C[i][j];
}
m/=2;
}
printf("Values of X:\n");
for(i=0;i<n;i++)
printf("%.6f ",(A[i][c-1]/A[i][i]));
printf("\n");
printf("\nTime taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
exit(0);}
```

<u>Parallelized code (using 5 threads):</u>

```
#include <bits/stdc++.h>
#include <pthread.h>
#include <time.h>
#include <semaphore.h>
#include <stdlib.h>
#define ll int
#define pb push_back
#define r 200
#define c 201
using namespace std;
float A[r][c],B[r][c],C[r][c];
ll l=0,n=r,m=n,t,x,ti,i,j,k,q;
pthread_t ptd[5];

void *soln(void *arg){
if(m>1 && x<(q-1)){
t=0;
for(i=0;i<r;i++){
for(j=0;j<c;j++){
B[i][j]=A[i][j];C[i][j]=A[i][j];}}

for(t=0;t<(n/m);t++){
for(k=0+t*m;k<((m/2)+t*m);k++){
float max=B[k][k];
for(i=k;i<m+t*m;i++)
if(B[i][k]>=max){max=B[i][k];q=i;}
for(i=0+t*m;i<m+t*m;i++)
{float g=B[q][i];B[q][i]=B[k][i];B[k][i]=g;}
for(q=k+1;q<m+t*m;q++)
B[q][k]=(B[q][k]/B[k][k]);
for(j=k+1;j<m+t*m;j++)
for(i=k+1;i<m+t*m;i++)
```

```
B[i][j]-=((B[i][k])*(B[k][j]));
for(i=k+1;i<m+t*m;i++)
B[i][c-1]-=((B[i][k])*(B[k][c-1]));
}

for(i=0+t*m;i<((m/2)+t*m);i++)
for(j=0+t*m;j<((m/2)+t*m);j++)
A[i][j]=B[i][j];
for(i=0+t*m;i<((m/2)+t*m);i++)
A[i][c-1]=B[i][c-1];
}

for(t=0;t<(n/m);t++){
for(k=c-2-t*m;k>=(c-2)-(m/2)-t*m+1;k--){
float max=C[k][k];
for(i=k;i>=t*m;i--)
if(C[i][k]>=max){max=C[i][k];q=i;}
for(i=(c-2)-t*m;i>=(c-2)-(t)*m-m+1;i--)
{float g=C[q+1][i];C[q+1][i]=C[k][i];C[k][i]=g;}
for(q=k-1;q>=(r-1)-t*m-m+1;q--)
C[q][k]=(C[q][k]/C[k][k]);
for(j=k-1;j>=(c-2)-(t)*m-m+1;j--)
for(i=k-1;i>=(r-1)-t*m-m+1;i--)
C[i][j]-=((C[i][k])*(C[k][j]));
for(i=k-1;i>=(r-1)-t*m-m+1;i--)
C[i][c-1]-=((C[i][k])*(C[k][c-1]));
}
for(i=c-2-t*m;i>=(c-2)-(m/2)-t*m+1;i--)
for(j=c-2-t*m;j>=(c-2)-(m/2)-t*m+1;j--)
A[i][j]=C[i][j];
for(i=c-2-t*m;i>=(c-2)-(m/2)-t*m+1;i--)
A[i][j]=C[i][j];
}
m/=2;}
else{
while(m>1){
t=0;
for(i=0;i<r;i++){
for(j=0;j<c;j++){
B[i][j]=A[i][j];C[i][j]=A[i][j];}}

for(t=0;t<(n/m);t++){
for(k=0+t*m;k<((m/2)+t*m);k++){
float max=B[k][k];
for(i=k;i<m+t*m;i++)
if(B[i][k]>=max){max=B[i][k];q=i;}
for(i=0+t*m;i<m+t*m;i++)
{float g=B[q][i];B[q][i]=B[k][i];B[k][i]=g;}
for(q=k+1;q<m+t*m;q++)
B[q][k]=(B[q][k]/B[k][k]);
for(j=k+1;j<m+t*m;j++)
for(i=k+1;i<m+t*m;i++)
B[i][j]-=((B[i][k])*(B[k][j]));
for(i=k+1;i<m+t*m;i++)
B[i][c-1]-=((B[i][k])*(B[k][c-1]));
}
```

```
for(i=0+t*m;i<((m/2)+t*m);i++)
for(j=0+t*m;j<((m/2)+t*m);j++)
A[i][j]=B[i][j];
for(i=0+t*m;i<((m/2)+t*m);i++)
A[i][c-1]=B[i][c-1];
}

for(t=0;t<(n/m);t++){
for(k=c-2-t*m;k>=(c-2)-(m/2)-t*m+1;k--){
float max=C[k][k];
for(i=k;i>=t*m;i--)
if(C[i][k]>=max){max=C[i][k];q=i;}
for(i=(c-2)-t*m;i>=(c-2)-(t)*m-m+1;i--)
{float g=C[q+1][i];C[q+1][i]=C[k][i];C[k][i]=g;}
for(q=k-1;q>=(r-1)-t*m-m+1;q--)
C[q][k]=(C[q][k]/C[k][k]);
for(j=k-1;j>=(c-2)-(t)*m-m+1;j--)
for(i=k-1;i>=(r-1)-t*m-m+1;i--)
C[i][j]-=((C[i][k])*(C[k][j]));
for(i=k-1;i>=(r-1)-t*m-m+1;i--)
C[i][c-1]-=((C[i][k])*(C[k][c-1]));
}
for(i=c-2-t*m;i>=(c-2)-(m/2)-t*m+1;i--)
for(j=c-2-t*m;j>=(c-2)-(m/2)-t*m+1;j--)
A[i][j]=C[i][j];
for(i=c-2-t*m;i>=(c-2)-(m/2)-t*m+1;i--)
A[i][j]=C[i][j];
}
m/=2;}
}}

int main()
{int q;
printf("Enter no. of threads:");
scanf("%d",&q);
clock_t tStart = clock();
for(i=0;i<r;i++){
for(j=0;j<c;j++){
A[i][j]=rand()%25;
}}
for(x=0;x<q;x++)
pthread_create(&ptd[x],NULL,soln,&x);
for(i=0;i<q;i++)
pthread_join(ptd[i],NULL);
printf("Values of X:\n");
for(i=0;i<n;i++)
printf("%.6f ",(A[i][c-1]/A[i][i]));
printf("\n");
printf("\nTime taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
exit(0);}
```

Quite significant performance gain was observed in the running time of the parallelized version of the iterative solution of SGE in compared to that of the recursive solution. The results are tabulated below.

| **Order/Method** | 25 | 100 | 200 | 500 | 800 | 900 | 1000 |
|---|---|---|---|---|---|---|---|
| **Recursive (sec)** | 0.00283 | 0.043988 | 0.296094 | 4.534074 | 19.710256 | 28.365654 | 39.814362 |
| **Iterative(sec)** | 0.000791 | 0.021037 | 0.156625 | 2.189652 | 11.239122 | 16.320862 | 26.467252 |

- **Multiplication of 2 matrices using static as well as dynamic allocation of threads:** We have to multiply 2 sparse matrices using static as well as dynamic allocation of threads.

Static allocation of threads: In this method, I have explicitly assigned the task of each thread before runtime (hence static).

Parallelized code (using 20 threads):

```
#include<stdio.h>
#include<bits/stdc++.h>
#include<pthread.h>
#include<unistd.h>
#include<stdlib.h>
#include<mutex>
#include <semaphore.h>
using namespace std;
int r1=MAX,r2=MAX,c1=MAX,c2=MAX,num=0,num_threads=5;
vector<vector<long long int> > mat1,mat2,res_mat;
mutex m;
sem_t mu;
void *mult_thread(void *param)
{
            int i,tot=0;
            int row_num=*((int *)param);
          int k=row_num;
            while( k < r1){
            for(int j=0;j<c2;j++)
          {
              tot=0;
               for(i=0;i<c1;i++)
                  {
                        tot += mat1[k][i]*mat2[i][j];
                  }

               res_mat[k][j]=tot;
          }

         k+=num_threads;
       }
}
void accept(vector<vector<long long int> > &mat, int r,int c)
{
            long long int val=0;

            for(int i=0;i<r;i++){
               vector<long long int>v;
                  for(int j=0;j<c;j++)
                        {
```

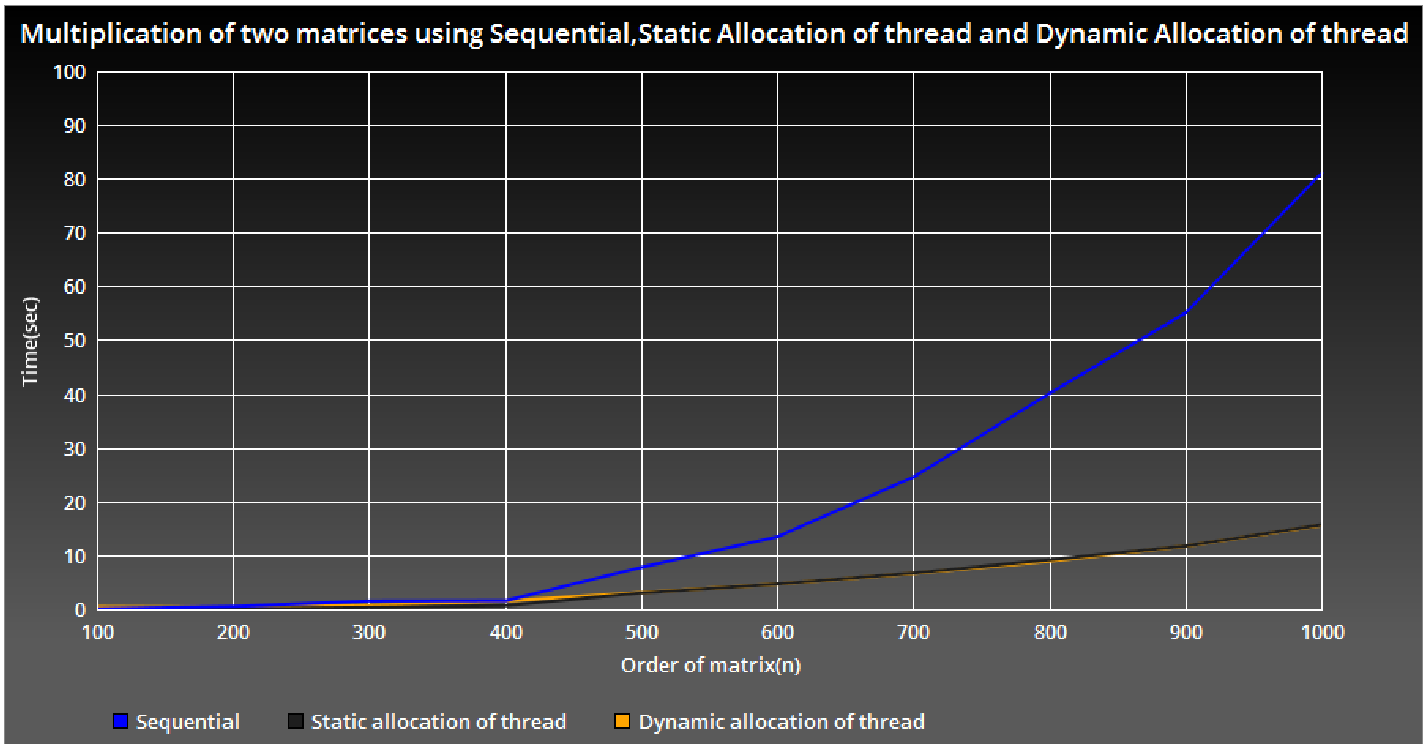
```
                        v.push_back(val++);
                 }
                   mat.push_back(v);
           }
}
void display()
      { for(int i=0;i<r1;i++)
              {
                for(int j=0;j<c2;j++)
                   {
                    printf("%d ",res_mat[i][j]);
                   }
                  cout<<endl;
       }}

int main()
{
        clock_t tStart = clock();
           int i,j;
           accept(mat1,r1,c1);
           accept(mat2,r2,c2);
           for(int i=0;i<r1;i++)
            {  vector<long long int>v;
              for(int j=0;j<c2;j++)
               v.push_back(0);
           res_mat.push_back(v);
           }
         pthread_t p[num_threads];

           void *exit_status;
           sem_init(&mu,0,1);
           if(c1!=r2)
           cout<<"Invalid size of matrix for multiplication"<<endl;

           else if(c1==r2)
                 {
                   int row[num_threads];
                   int i=0;
                       while(i<num_threads)
                             {
                                row[i]=i;

      pthread_create(&p[i],NULL,*mult_thread,&row[i]);
                                ++i;
                             }
                 }
        for(i=0;i<num_threads;i++)pthread_join(p[i],&exit_status);
           printf("\nTime taken:%.6fs\n",(clock() - tStart)/CLOCKS_PER_SEC);
           display();
           return 0;
      }
```

<u>Dynamic allocation of threads:</u> In this method the task of each thread is implicitly assigned by the kernel. It is unknown to us before runtime and is decided during the runtime (hence dynamic).

Parallelized code (using 20 threads):

```
#include<stdio.h>
#include<bits/stdc++.h>
#include<pthread.h>
#include<unistd.h>
#include<stdlib.h>
#include<mutex>
#include <semaphore.h>
using namespace std;
#define MAX 10
int r1=MAX,r2=MAX,c1=MAX,c2=MAX,num,num_threads=20;
vector<vector<long long int> > mat1,mat2,res_mat;
mutex m;
sem_t mu;
void *mult_thread(void *param)
{
            int i,tot=0;
            int row_num=*((int *)param);
          int k=row_num;
            while( k < r1){
            for(int j=0;j<c2;j++)
          {
              tot=0;
               for(i=0;i<c1;i++)
                  {
                        tot += mat1[k][i]*mat2[i][j];
                  }

               res_mat[k][j]=tot;
          }
          sem_wait(&mu);
           ++num;
         k=num;
        sem_post(&mu);
       }}

void accept(vector<vector<long long int> > &mat, int r,int c)
{
            long long int val=0;

            for(int i=0;i<r;i++){
               vector<long long int>v;
                  for(int j=0;j<c;j++)
                        {
                        v.push_back(val++);
                 }
                   mat.push_back(v);
           }}

void display()
      {for(int i=0;i<r1;i++)
              {
                for(int j=0;j<c2;j++)
                   {
                    printf("%d ",res_mat[i][j]);
                   }
                  cout<<endl;
             }}

int main()
{
        clock_t tStart = clock();
            int i,j;
            accept(mat1,r1,c1);
            accept(mat2,r2,c2);
            for(int i=0;i<r1;i++)
             {
               vector<long long int>v;
               for(int j=0;j<c2;j++)
                v.push_back(0);
            res_mat.push_back(v);
            }
          pthread_t p[num_threads];
            void *exit_status;
            sem_init(&mu,0,1);
            if(c1!=r2)
            cout<<"Invalid size of matrix for multiplication"<<endl;
            else if(c1==r2)
                  {
                    num=num_threads-1;
                    int row[num_threads];
                    int i=0;
                        while(i<num_threads)
                              {
                                 row[i]=i;

       pthread_create(&p[i],NULL,*mult_thread,&row[i]);
                                 ++i;
                              }
                  }
        for(i=0;i<num_threads;i++)pthread_join(p[i],&exit_status);
            printf("\nTime taken: %.6fs\n",clock() - tStart)/CLOCKS_PER_SEC);
            display();
            return 0;
      }
```

A graph was plotted using the observed experimental data of the above two programs and was compared with the sequential one. It was observed that the static & dynamic thread allocation program took approximately same running time but that was much less in comparison to the sequential matrix multiplication of 2 matrices.



- **<u>Efficient chain matrices multiplication through dynamic programming using multi-threading:</u>** This problem involves the question of determining the optimal sequence for performing a series of operations. This general class of problem is important in compiler design for code optimization and in databases for query optimization.

  Matrix multiplication is an associative but not a commutative operation. This means that we are free to parenthesize the above multiplication however we like, but we are not free to rearrange the order of the matrices. Consequently, this problem reduces to determining the order of multiplication that minimizes the number of operations, that was performed using dynamic programming.

The sequential as well as a parallelized version of the problem was made and experimental results were analyzed.

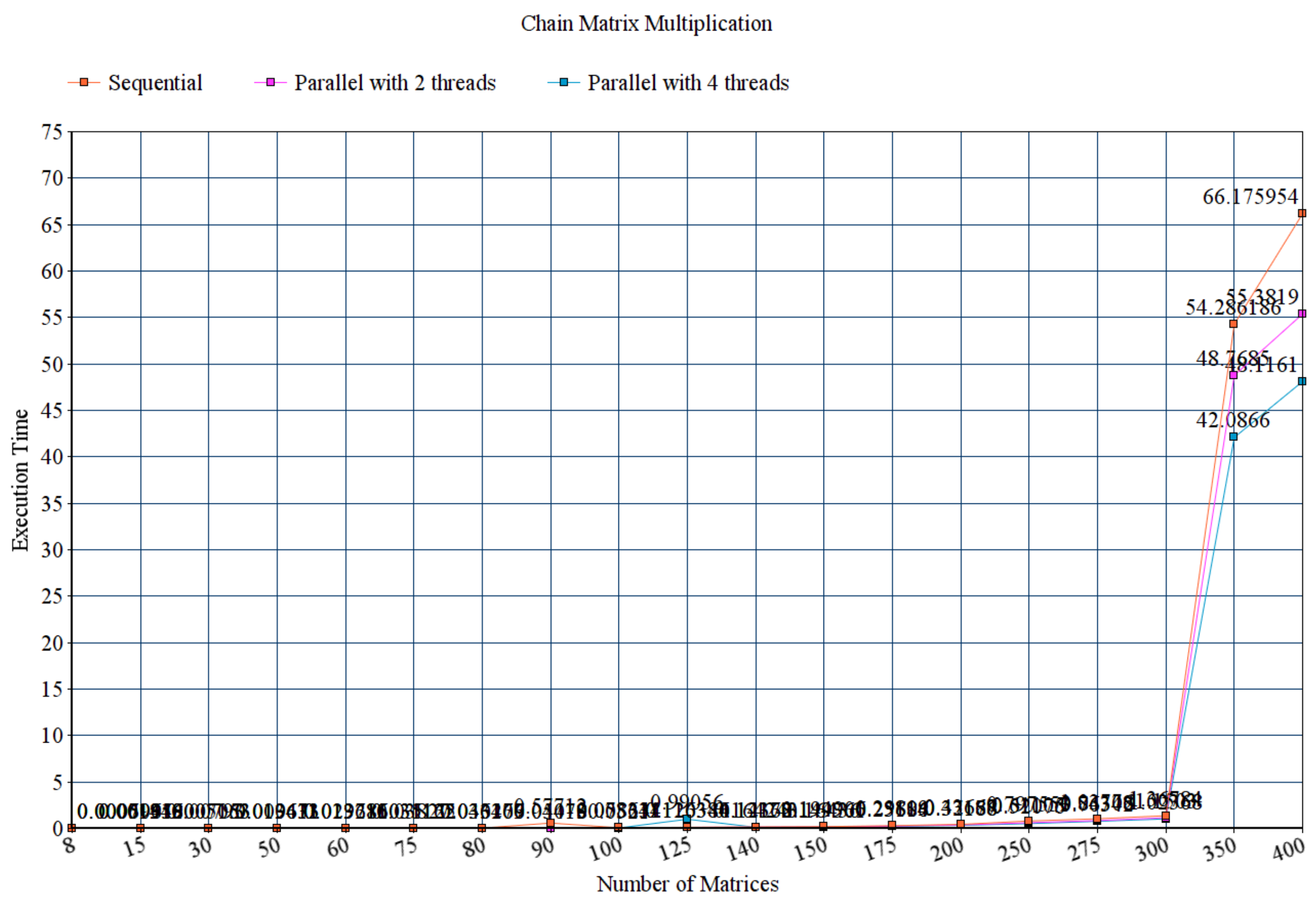
<u>Sequential code:</u>

```
#include <bits/stdc++.h>
#define ll long long int
#define pb push_back
#define mp make_pair
using namespace std;
int n;//no. of matrices
int s[500][500],m[500][500];
vector<int>p;
vector<vector<int> >A[500];

void create(){int d;
        for(int i=1;i<=n;i++){
                int x=p[i-1],y=p[i];vector<vector<int> > W;
                        for(int t=0;t<x;t++){vector<int> v;
                                for(int l=0;l<y;l++)
                                        {d=rand()%4;v.pb(d);}
                                W.pb(v);
                        }
          A[i]=(W);
        }
}

void match(vector<int> r){
        for(int i=1;i<=n;i++)m[i][i]=0;
                for(int l=2;l<=n;l++){
                        for(int i=1;i<=n-l+1;i++){
                                int j=i+l-1;m[i][j]=INT_MAX;
                                for(int k=i;k<=j-1;k++){
                                 int q=m[i][k]+m[k+1][j]+r[i-1]*r[k]*r[j];
                                        if(q<m[i][j]){
                                                m[i][j]=q;s[i][j]=k;
                                        }
                                }
                        }
                }
}

vector<vector<int> > mul(vector<vector<int> > X,vector<vector<int> > Y){
        int m=X.size(),n=X[0].size(),k=Y[0].size();
        vector<vector<int> > Z;
        for(int i=0;i<m;i++){
                vector<int> v;
                for(int j=0;j<k;j++){int d=0;
                        for(int p=0;p<n;p++){
                                d+=X[i][p]*Y[p][j];
                        }
                        v.pb(d);
                }
                Z.pb(v);
        }
        return Z;
}

vector<vector<int> > chmul(int i,int j){
        if(i==j)return A[i];
        else{
                int k=s[i][j];
                vector<vector<int> > X=chmul(i,k);
                vector<vector<int> > Y=chmul(k+1,j);
                return mul(X,Y);
        }
}

int main(){
        scanf("%d",&n);p.assign(n+1,0);
        for(int i=0;i<=n;i++)p[i]=i+2;
                clock_t tStart = clock();
        create();match(p);
        int x=p[0],y=p[n];vector<vector<int> > P;
        P=chmul(1,n);
        for(int i=0;i<x;i++){
                for(int j=0;j<y;j++)
                        printf("%d ",P[i][j]);
                printf("\n");
        }
        printf("\nTime taken: %.6fs\n",(clock() - tStart)/CLOCKS_PER_SEC);
        return 0;
}
```

## Parallelized code (with no. of threads as I/P parameter):

```
#include <bits/stdc++.h>
#include <pthread.h>
#include <semaphore.h>
#include <mutex>
#define ll long long int
#define pb push_back
#define mp make_pair
using namespace std;
int n,u,q=0,w=1;//no. of matrices
int s[500][500],m[500][500];
vector<int>p;
vector<vector<int> >A[500];
mutex mu;
sem_t z;

void *create(void *arg){
lock_guard<mutex>guard(mu);
if(w<u-1){int d;
                int x=p[w-1],y=p[w];vector<vector<int> > W;
                        for(int t=0;t<x;t++){vector<int> v;
                                for(int l=0;l<y;l++)
                                        {d=rand()%4;v.pb(d);}
                                        W.pb(v);
                        }
                        A[w]=(W);
                        sem_wait(&z);
                        w++;
                        sem_post(&z);
        }
        else{   for(int i=w;i<=n;i++){
                int x=p[i-1],y=p[i];vector<vector<int> > W;
                        for(int t=0;t<x;t++){vector<int> v;
                                for(int l=0;l<y;l++)
                                        {int d=rand()%4;v.pb(d);}
                                        W.pb(v);
                        }
                        A[i]=(W);
        }
        }}

void match(vector<int> r){
        for(int i=1;i<=n;i++)m[i][i]=0;
                for(int l=2;l<=n;l++){
                        for(int i=1;i<=n-l+1;i++){
                                int j=i+l-1;m[i][j]=INT_MAX;
                                for(int k=i;k<=j-1;k++){
                                int q=m[i][k]+m[k+1][j]+r[i-1]*r[k]*r[j];
                                        if(q<m[i][j]){
                                                m[i][j]=q;s[i][j]=k;
                                }
                        }
                }
        }
}

vector<vector<int> > mul(vector<vector<int> > X,vector<vector<int> > Y){
        int m=X.size(),n=X[0].size(),k=Y[0].size();
        vector<vector<int> > Z;
        for(int i=0;i<m;i++){
                vector<int> v;
                for(int j=0;j<k;j++){int d=0;
                        for(int p=0;p<n;p++){
                                d+=X[i][p]*Y[p][j];
                        }
                        v.pb(d);
                }
                Z.pb(v);
        }
        return Z;
}

vector<vector<int> > chmul(int i,int j){
        if(i==j)return A[i];
        else{
                int k=s[i][j];
                vector<vector<int> > X=chmul(i,k);
                vector<vector<int> > Y=chmul(k+1,j);
                return mul(X,Y);
        }
}

int main(){
        cout<<"Enter number of threads :";
     cin>>u;
        scanf("%d",&n);p.assign(n+1,0);
        for(int i=0;i<=n;i++)p[i]=i+2;
                clock_t tStart = clock();
        pthread_t mat[u];
        void *exit_status;
     sem_init(&z,0,1);
     while(q<u){
        pthread_create(&mat[q],NULL,create,NULL);
        ++q;}
      for(int i=0;i<u;i++)
      pthread_join(mat[i],&exit_status);
        match(p);
        int x=p[0],y=p[n];vector<vector<int> > P;
        P=chmul(1,n);
        for(int i=0;i<x;i++){
                for(int j=0;j<y;j++)
                        printf("%d ",P[i][j]);
                printf("\n");
        }
        printf("\nTime taken: %.6fs\n",(clock() - tStart)/CLOCKS_PER_SEC);
        return 0;}
```

The parallelized version was executed with 4 threads (also with 2 threads), resulting in a running time of 42.0866 seconds (48.7685 seconds using 2 threads) in comparison to the sequential running time of 54.286186 seconds, for the chain matrix multiplication of 350 matrices. For 400 matrices, the 4 threaded parallelized version executed in 48.1161 seconds (48.7685 seconds for 2 threaded version),while the sequential version executed in 66.175954 seconds.
A comparison of running times is drawn in the form of following graph.



## REFERENCES:

- K.N. Balasubramanya Murthy, Srinivas Aluru and S. Kamal Abdali, *Solving Linear Systems on Linear Processor Arrays using a ⋆-Semiring Based Algorithm*
- K.N. Balasubramanya Murthy and C. Siva Ram Murthy, *A New Gaussian Elim--ination-Based Algorithm for Parallel Solution of Linear Equations, Computers Math. Applic. Vol. 29, No. 7, pp. 39-54, 1995*
- D. Tang and G. Gupta, *An Efficient Parallel Dynamic Programming Algorithm, Computers Math. Applic. Vol. 30, No. 8, pp. 65-74, 1995, Pergamon.*