Indian Institute of Information Technology, Vadodara

Parallel Programming(CS403)

Prof. Reshmi Mitra

**Name** - Dilip Puri
**ID** - 201351014
**Collaborators** - Hemant Kumar(201352026)
Govind Meena(201352010)

---

**Project 1**

---

**Submission Date** - November 11, 2016 **Deadline** - Nov. 11, 11.59 PM

---

# 1 Introduction

Suppose we have the system of equations

$$AX=B$$

Even most of the mathematical model follows these kind of systems. The motivation for an LU decomposition is based on the observation that systems of equations involving triangular coefficient matrices are easier to deal with. Indeed, the whole point of Gaussian Elimination is to replace the coefficient matrix with one that is triangular. The LU decomposition is another approach designed to exploit triangular systems. We suppose that we can write

$$A = LU$$

where L is a lower triangular matrix and U is an upper triangular matrix. Our aim is to find L and U and once we have done so we have found an LU decomposition of A. Let A be a square matrix. If there is a lower triangular matrix L with all diagonal entries equal to 1 and an upper matrix U such that A=LU, then we say that A has an LU-decomposition. It can be helpful in calculating various types of operation on matrices. Here L matrix has the upper triangular values as 0 and U has lower triangular values as 0 and diagonal values same as that of the original matrix.

# 2 Algorithm

To decompose matrix A into L and U we are using Gaussian Elimination method which is described as follows

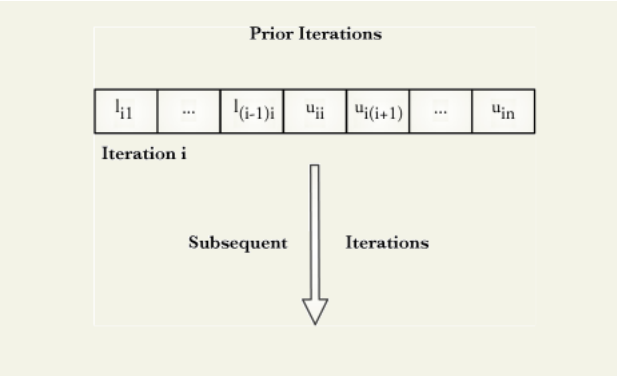


Figure 1: Computational Sequence of Method

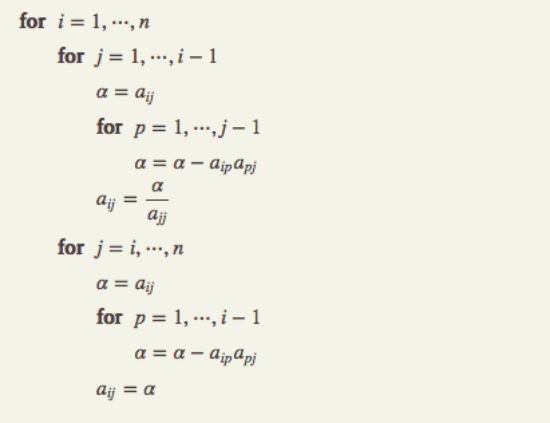


Figure 2: LU Decomposition Algorithm

# 3 Dependency Graph

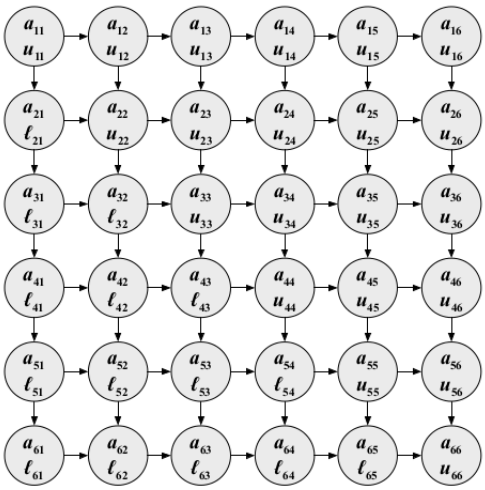

Figure 3: Dependency Graph

# 4 Parallel Code

Listing 1: Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
#include <time.h>

int size;
double **l;
double **u;
double **mat;
double **matcpy;
clock_t begin, end;
FILE *fp;

void printMatrix(double **m) {
    printf("\n \n");

   int i =0, j=0;
    for (i=0;i<size;i++) {
        for(j=0;j<size;j++) {
            printf("| %f  | ",m[i][j]);
        }
        printf("\n");
    }
}

void freeMatrix(double **matrix)
{
   int i =0;
    for(i = 0; i < size; i++)
        free(matrix[i]);
    free(matrix);
}


int decompose(double **m) {
   begin =clock();
   int i=0, j=0, k=0;
   double factor;

#pragma omp for
     for(j =0; j < size-1; j++) {
       for(i = j+1; i < size; i++) {
           factor = m[i][j]/m[j][j];
           for(k = 0; k < size; k++) {
              u[i][k]= m[i][k] - (m[j][k] * factor);
           }
           l[i][j] = factor;
       }
       //matrix copy m = u
       for(i =0; i< size; i++) {
        for(k=0; k < size; k++) {
```

```
            m[i][k] = u[i][k];
         }
        }
        //copy end
    }

    for(i = 0; i < size; i++) {
        l[i][i] =1;
    }
    end = clock();
    return 1;
}

int generateMatrix(int size)
{
    srand(time(NULL));
    mat = malloc(size * (sizeof *mat));
    u = malloc(size * (sizeof *u));
    l = malloc(size * (sizeof *l));
    //matcpy = malloc(size * (sizeof *matcpy));

    int i =0;
    for(i = 0; i<size;i++)
    {
      u[i] = malloc((sizeof *u[i]) * size);
      l[i] = malloc((sizeof *l[i]) * size);
      mat[i] = malloc((sizeof *mat[i]) * size);
    }
    for( i=0 ; i < size ; i++) {
      u[i] = malloc((sizeof *u[i]) * size);
      l[i] = malloc((sizeof *l[i]) * size);
        mat[i] = malloc((sizeof *mat[i]) * size);
        //matcpy[i] = malloc((sizeof *matcpy[i]) * size);

        int j = i;
        for( j=i; j< size; j++) {
            mat[i][j] = i+1;
            mat[j][i] = i+1;
            u[i][j] = mat[i][j];
            u[j][i] = mat[j][i];
        }
    }
    return 1;
}


int main(int argc, char * argv[])
{
    if(argc < 2) {
      printf("Matrix Size missing in the arguments \n");
      return -1;
    }

    size = strtol(argv[1],(char **)NULL,10);
    fp = fopen("output.txt","a+");
```

```
    generateMatrix(size);
   //setting number of threads
   int nthreads = 32;
   int s = 2;

   for(s=2 ; s <= nthreads; s= s*2)
   {
      omp_set_num_threads(s);
      omp_set_nested(1);

      int successFlag = 0;
      if(decompose(mat) == 1)
      {
         successFlag = 1;
      }
      printf("u matrix \n");
      printMatrix(u);
      printf("l matrix \n");
      printMatrix(l);



      int current_threads = omp_get_num_threads();

      fprintf(fp,"%d ", size);
      fprintf(fp,"%d ",s);
      fprintf(fp,"%f\n", (double)(end - begin) / CLOCKS_PER_SEC);

   }
   fclose(fp);
    return 0;
}
```

# 5 Performance Graph

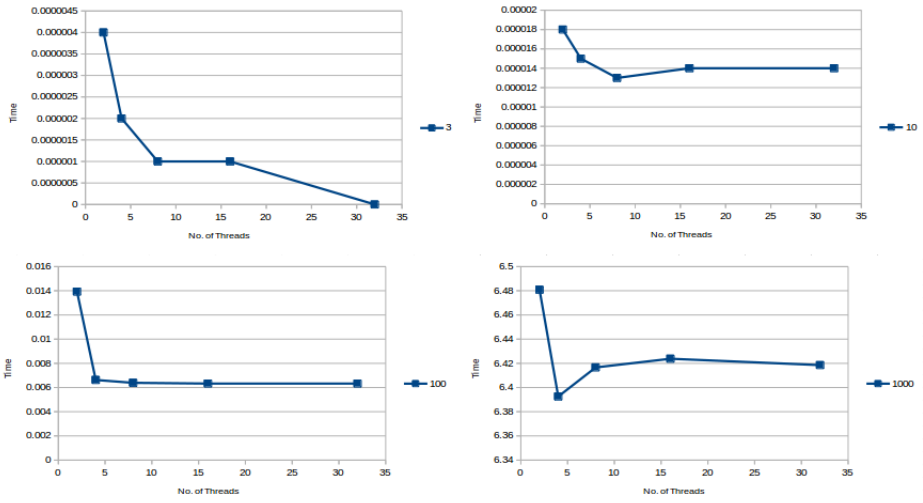


Figure 4: Performance

# 6 SpeedUP calculation

| Matrix Size\#of Threads | 3 | 10 | 100 | 1000 |
|---|---|---|---|---|
| 2 | 1 | 1 | 1.04157942 | 1.000135957 |
| 4 | 0.5 | 0.8333333333 | 0.495438229 | 0.986511664 |
| 8 | 0.25 | 0.7222222222 | 0.477714628 | 0.990224953 |
| 16 | 0.25 | 0.7777777778 | 0.473302423 | 0.991333446 |
| 32 | 0 | 0.7777777778 | 0.473302423 | 0.990521405 |

Figure 5: Speed Up Table

# 7 CPU usage

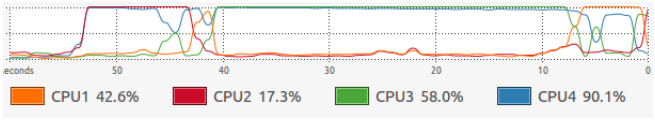


Figure 6: For 2 threads

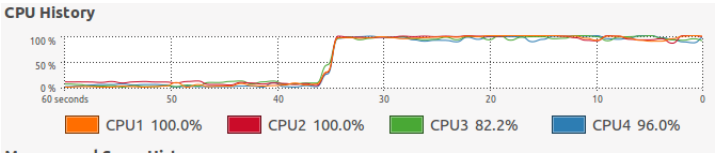


Figure 7: For 4 threads

# 8 Analysis using Callgrind and Cache misses calculation

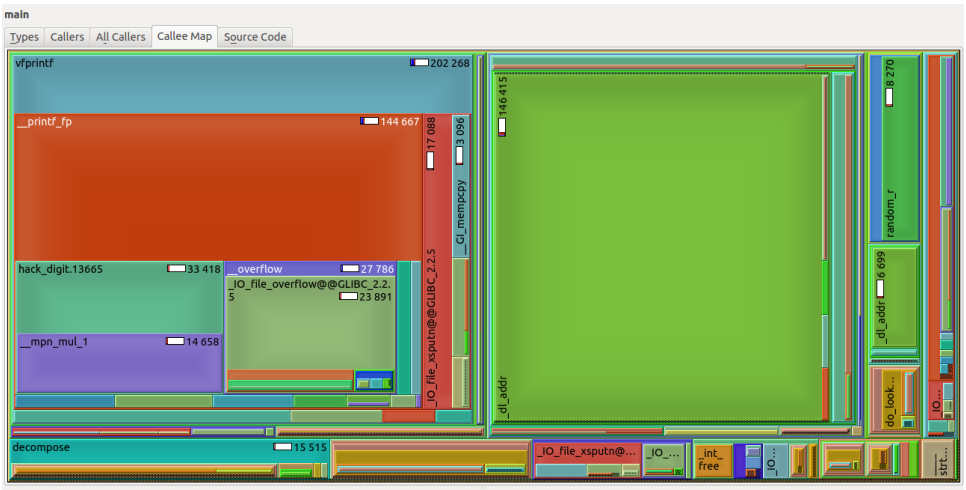


Figure 8: Callee Map for parallel code

```
==5581==
==5581== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==5581== Collected : 501008 125852 65614 1323 3200 778 1297 2259 687
==5581==
==5581== I   refs:      501,008
==5581== I1  misses:      1,323
==5581== LLi misses:      1,297
==5581== I1  miss rate:    0.26%
==5581== LLi miss rate:    0.25%
==5581==
==5581== D   refs:      191,466  (125,852 rd + 65,614 wr)
==5581== D1  misses:      3,978  (  3,200 rd +    778 wr)
==5581== LLd misses:      2,946  (  2,259 rd +    687 wr)
==5581== D1  miss rate:     2.0% (    2.5%   +    1.1%  )
==5581== LLd miss rate:     1.5% (    1.7%   +    1.0%  )
==5581==
==5581== LL refs:         5,301  (  4,523 rd +    778 wr)
==5581== LL misses:       4,243  (  3,556 rd +    687 wr)
==5581== LL miss rate:      0.6% (    0.5%   +    1.0%  )
```

Figure 9: Cache misses for parallel code