Workload flow calculation based on linear programming: code development and performance evaluation

by Pascal CHEN.

date : 2nd July 2019.

This work was developed in the *University of Las Palmas de Gran Canaria, SIANI research institute*, Spain, during an internship stage approved by: *Polytech Sorbonne of the Sorbonne Universite*, France.

Contents

1 ABSTRACT

This work was done in order to solve a problem: the problem of balancing tasks for a partitioning knowing that we have a first result, under constraints that each partition is strongly related to its neighbors.
For example, in the case of the mesh, it requires a lot of calculations, which is why we partition and the first approach to partitioning is to cut the mesh at equal size, but the workload is not necessarily even-handed. Also, to balance the tasks, the transfer can only be done between neighbors, a calculation does not make sense on two separate meshes, that's why we will use the simplex method to obtain the minimum of exchange to rebalance the tasks.

2 INTRODUCTION

2.1 Linear programming

Linear programming also called linear optimization is a method to achieve one type of problem :

$$\text{Maximize } c^T x$$
$$\text{subject to } Ax \leq b$$
$$\text{and } x \geq 0.$$

where x represents the vector of variables, c and b are vectors of coefficients, A is a matrix of coefficients.
For more details :
- https://en.wikipedia.org/wiki/Linear_programming

2.2 Simplex method

Simplex method is a popular method for linear programming. The simplex method can be seen as a displacement on the vertices of a polytope, it begins at a starting vertex and moves along the edges of the polytope until it reaches the vertex of the optimal solution. Each iteration must improve the objective function, which is why convexity is important because it means that there is always this accessible path to reach the optimal solution.
For more details :
- https://en.wikipedia.org/wiki/Simplex_algorithm

2.3 Library

There are many libraries that deal with this problem, to quote one of them, there is glpk (gnu linear programming kit).
For more details:
- https://www.gnu.org/software/glpk/
- https://en.wikipedia.org/wiki/GNU_Linear_Programming_Kit

3 PROBLEM STATEMENT

3.1 Parallel computing

In this part, I will introduce the problem of partitioning.

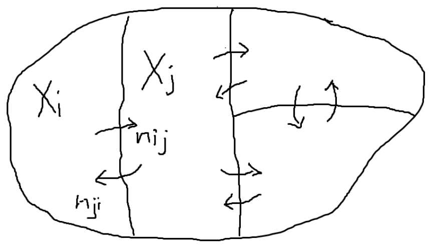


Fig. 1. Partitioning.

The context is the following: A work is divided into several subtasks, in order to parallelize all this and to make the work as quickly as possible.

3.2 Mesh partitioning

However some tasks are interconnected by a boundary, ie a partition is very strongly related to its neighbors, for example in the case of the mesh.

If we cut a mesh so that each partition performs a calculation on it, we will cut so that each mesh is a united block and not pieces of meshes scattered.

3.3 Measure the workload after parallel execution

Here is the problem: we have N partitions where each partition has a workload, but the workload is different for partitions, so some computer cores are forced to wait for their neighbors to complete the program.

3.4 Load balancing

We will therefore seek to find a way to balance the workload with the least possible exchange between neighbors so that the exchange is done as quickly as possible.

3.5 Our solution: workload exchange calculation based on linear programming

This is therefore a linear optimization problem where the number of workload exchanges must be minimized.
It can be modeled as follows:

Let i and j be in $[1; N]$.
Let N_i be the physical workload of partition number i
(for example: time spent executing a task)
and X_i the normalized workload of N_i, ie

$$X_i = N_i - \tilde{N}$$

where :

$$\tilde{N} = \frac{1}{N} \sum_{i=1}^{N} N_i.$$

Let n_{ij} the workload exchange from i to j.
Note that $\forall i$, $n_{ii} = 0$.
Let $\mathcal{N}(i)$ the set of neighbors of i.
Note that $\forall i, \forall j \notin \mathcal{N}(i)$, $n_{ij} = 0$.

$$\text{Minimize} \sum_{i,j} n_{ij}$$
$$\text{subject to } \forall i, X_i - \sum_j n_{ij} + \sum_j n_{ji} = 0.$$

An equivalent algebraic view is as follow:

Let i and j be in $[1; N]$.
Let N_i be the physical workload of partition number i
and X_i the normalized workload of N_i, ie

$$X_i = N_i - \tilde{N}$$

Let n_{ij} the workload exchange from i to j.
Let B the boundary matrix where b_{ij} is equals to 1 if $j \in \mathcal{N}(i)$, or equals to 0.

$$\text{Minimize} \sum_{i,j} n_{ij} b_{ij}$$
$$\text{subject to } \forall i, X_i - \sum_j n_{ij} bij + \sum_j n_{ji} b_{ji} = 0.$$

Note that

$$\sum_i X_i = 0$$

and one of the equations is redundant in the constraints.

4 EXPERIMENTAL METHODOLOGY

Table 1. The main characteristics of the server computer used

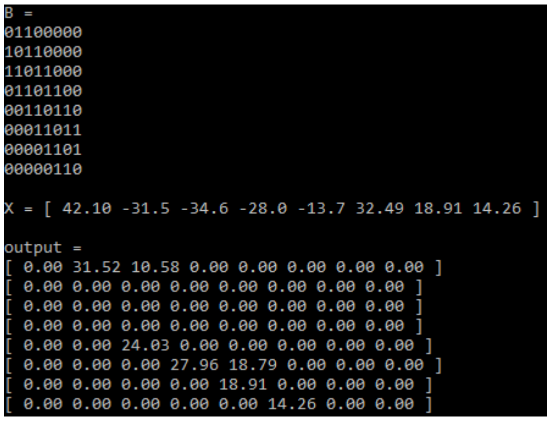
Name	Value
Nb of cores	8
CPU name	Intel(R) Xeon(R) CPU E3-1270 v5
Frequency	3.6 GHz
Cache	8 MB
Bus speed	8 GT/s DMI3
Instruction set	64-bit

Table 2. The versions used

Name	Value
OS	Ubuntu 14.04.4
Compiler	gcc v4.8.4
lib	glpk v4.52

5 RESULTS

Of course, a constraint equation can be seen as two inequalities, but this is a trap for the simplex method, because you multiply the number of lines by two and add columns to the simplex matrix, whereas the method of the simplex is exponential although on average is polynomial.
Moreover if you run the simplex algorithm on inequalities to describe an equation, you will notice that the two lines will be very similar.
In this case, it is better to use two phases method.
I have tried to implement simplex method in C with dense matrix in order to compare with the library glpk (gnu linear programming kit).
here are the results for 8 partitions and maximum of 4 neighbors:

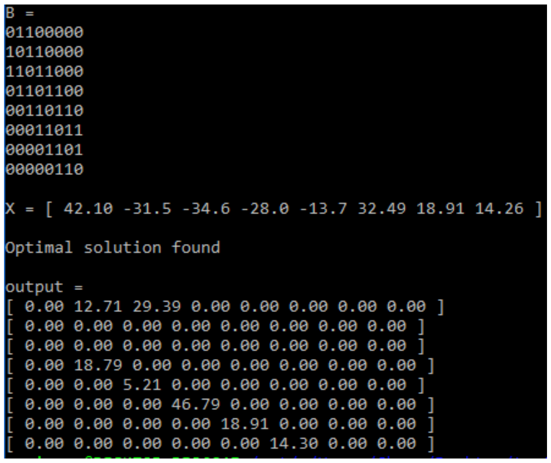


```
B =
01100000
10110000
11011000
01101100
00110110
00011011
00001101
00000110

X = [ 42.10 -31.5 -34.6 -28.0 -13.7 32.49 18.91 14.26 ]

output =
[ 0.00 31.52 10.58 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 24.03 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 27.96 18.79 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 0.00 18.91 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 0.00 0.00 14.26 0.00 0.00 ]
```

Fig. 2. Glpk results.



```
B =
01100000
10110000
11011000
01101100
00110110
00011011
00001101
00000110

X = [ 42.10 -31.5 -34.6 -28.0 -13.7 32.49 18.91 14.26 ]

Optimal solution found

output =
[ 0.00 12.71 29.39 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 18.79 0.00 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 5.21 0.00 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 46.79 0.00 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 0.00 18.91 0.00 0.00 0.00 ]
[ 0.00 0.00 0.00 0.00 0.00 14.30 0.00 0.00 ]
```

Fig. 3. My results.

The values are generated randomly, I have copied the X output for my X, and the result are both acceptable, the objective sum is equal to 146.05 for one part and 146.1 for the other part due to the imprecision.

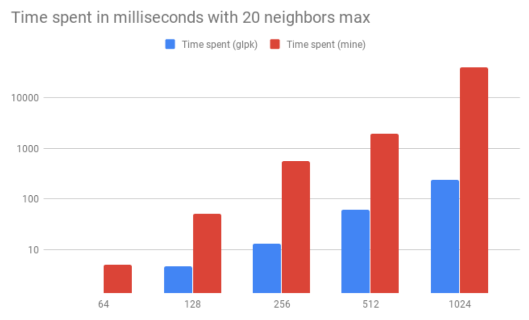


Fig. 4. Time comparison with 20 neighbors.

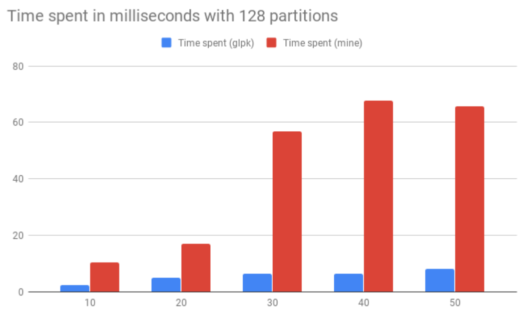


Fig. 5. Time comparison with 128 partitions.

Obviously, the library glpk is faster than mine.
These are the performance of glpk with a confidence interval of 5%:

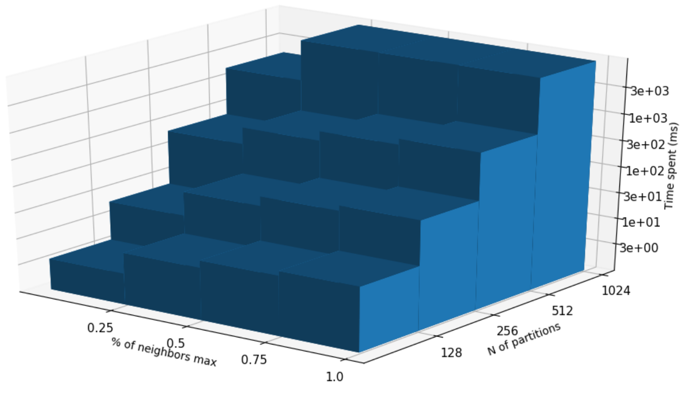


Fig. 6. Time spent in milliseconds for 100X% neighbors max and Y partitions.

In this graphic, the maximum means a segmentation fault due to the size of the matrix.

6 CONCLUSIONS

Due to the high performance of glpk, using sparse matrix, it is large better to use this library for memory and time complexity.
This kind of program can be linked to an input "ai" that can predict the workload.

7 REFERENCES

- https://en.wikipedia.org/wiki/Linear_programming
- https://en.wikipedia.org/wiki/Simplex_algorithm
- https://www.gnu.org/software/glpk/
- https://en.wikipedia.org/wiki/GNU_Linear_Programming_Kit

8 APPENDIX I

The simplex method use a table in order to solve the problem, here is the function which construct the table from B and n the size of B.

```c
double ** init_table(int ** B, int n, int nb_row, int nb_var) {
    int i, j, k = 0;
    double ** table = (double **) malloc(nb_row * sizeof(double *));
    for (i = 0 ; i < nb_row ; i ++) {
        table[i] = (double *) calloc(nb_var, sizeof(double));
    }
    for (i = 0 ; i < n ; i++) { // Main block
        for (j = 0 ; j < n ; j++) {
            if (B[i][j]) {
                if (i < n-1) {
                    table[i][k] = 1.0;
                }
                if (j < n-1) {
                    table[j][k] = -1.0;
                }
                k++;
            }
        }
    }
    return table;
}
```

But in order to construct this table, it is needed to know the size of the table: nb_row × nb_var.
The number of row is determined by the number of equation and the number of column is the number of variables.
Also, the library use sparse matrix, so it is needed to know the number of non-zero coefficients.

```c
/* count the number of variables and the number of non-zeros */
int nb_var = 0, nb_no0 = 0;
for (i = 0 ; i < n-1 ; i++) {
    for (j = i+1 ; j < n ; j++) {
        if (B[i][j]) {
            if (j < n-1) {
                nb_no0 += 2;
            } else {
                nb_no0++;
                nb_var++;
            }
        }
    }
}
nb_var += nb_no0;
nb_no0 <<= 1;

/* number of row */
int nb_row = n-1;
```

Then we have to fill the problem using the library:
- the first block represents the constraint of all rows have to be equal to $X[i]$
- the second block represents the constraint of all variables must be greater than 0 and the objective function is negative because we want to minimize it.

```c
/* fill problem */
char name[8];
glp_add_rows(lp, nb_row);
glp_add_cols(lp, nb_var);

for (i = 1 ; i <= nb_row ; i++) {
    sprintf(name, "%d", i);
    glp_set_row_name(lp, i, name);
    glp_set_row_bnds(lp, i, GLP_FX, X[i-1], X[i-1]);
}

for (i = 1 ; i <= nb_var ; i++) {
    sprintf(name, "x%d", i);
    glp_set_col_name(lp, i, name);
    glp_set_col_bnds(lp, i, GLP_LO, 0.0, 0.0);
    glp_set_obj_coef(lp, i, -1.0);
}
```

Then we have to fill the table, we have to skip all zeros.

```c
j = 0;
for (i = 1 ; i <= nb_no0 ; i++) {
    while (table[j/nb_var][j%nb_var] == 0)
    {
        j++;
        if (j == nb_row * nb_var) { break; }
    }
    if (j == nb_row * nb_var) { break; }
    ia[i] = j/nb_var+1, ja[i] = j%nb_var+1, ar[i] = table[j/nb_var][j%nb_var];
    j++;
}
```

Then launch the solver and get the output in the good format.

```c
double * x = (double *) calloc(nb_var, sizeof(double));
for (i = 0 ; i < nb_var ; i++) {
    x[i] = glp_get_col_prim(lp, i+1);
}

double ** out = (double **) malloc(n * sizeof(double *));
k = 0;
for (i = 0 ; i < n ; i++) {
    out[i] = (double *) calloc(n, sizeof(double));
    for (j = 0 ; j < n ; j++) {
        if (B[i][j]) {
            out[i][j] = x[k];
            k++;
        }
    }
}
```