Jacobi method for the solution of linear systems

Laura Fiorella 544419

July 12 2022

Abstract

The aim of this report is to summarize the main features of parallelized execution of a program using threads and Fast Flow tools for the implementation of Jacobi resolution method for linear system. We will focus on the main coding aspects of the various version of the algorithm and on the results obtained during the experimental phase. In particular we will analyse the effects of different choices of algorithmic parameters such as the size of the matrix, the parallel degree and the stopping criterion. We will refer to the formula of the method as:

$$x_i^{k+1} = \frac{1}{A_{ii}} b_i \sum_{\substack{j=1 \\ j\neq i}}^{n} A_{ij} x_j^k \tag{1}$$

Contents

1 Input generation

First of all we need to focus a little bit on the Jacobi method itself, in order to put us in the right hypothesis to guarantee the convergence of the method, and so the correctness of the results. The purpose is to solve a linear system: $Ax = b$, with A a generic n square matrix. Since the entire process involves a division by the diagonal elements of the matrix A, the first (and actually only) assumption that must hold is that A must be *row strictly diagonal dominant*:

$$\text{i.e} \quad \forall i \in \{1, ..., n\} \quad |A(i,i)| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |A(i,j)|$$

Under this hypothesis the method will converge to the solution of the linear system starting form every point $x_0 \in \mathbb{R}^n$. In the implementation phase the zero vector was used as starting point.
From a coding point of view we used a `std::vector<float>` to represent b and a `std::vector<std::vector<float»` for A, and for their generation we used the random `C++` function `std::rand()` which produces positive integers. As you can see in the code 1, after the generation of a single element of the matrix A, we scaled the number modulo 10 and then we divided it by 7 to obtain a `float` matrix with quite small number. The same operations were done to construct b. Then for the diagonal component of A, in order to obtain a row strictly diagonal dominant matrix, we impose that: $A[i][i] = 2 \times \sum_{\substack{j=1 \\ j\neq i}}^{n}$ `std::abs`$(A[i][j])$.

```
vector<vector<float>> matrixgen(int n){
        vector<vector<float>> M(n, vector<float>(n, 0));
        float sum=0;
        for (int i=0; i<n; i++){
                sum=0;
                for(int j=0; j<n; j++){
                        M[i][j]=(float(std::rand()%10))/7;
                        sum+=M[i][j];
                }
                M[i][i]=2*(sum-M[i][i]);
        return M;}
```

Listing 1: `matrixgen` code

2 Sequential algorithm

The first simple implementation of the Jacobi algorithm was done in a sequential way. The resulting code (1) is very simple to understand and does not need so much words to be spent on. It consists in three nested **for loops**, one on the number of iterations (N), one on the components of the new vector $x1$, which represent the result of each iteration of the method, and the last on the other horizontal components of the matrix A.
The only important consideration to be done regards this last loop: indeed, looking at the formula of the method (1), we can observe that for each row `i` of `x1` the variable `sum` must not contain the product `A[i][i]*b[i]`.
But checking the condition `j≠i` in each iteration would have been very expensive, so we decided to split the more internal `for` loop into two distinct loops, the first which goes from `0` to `i`, and the second which scans the indexes from `i+1` to `n`. The pseudocode of the the main operations of this sequential version of the algorithm is reported below.

Algorithm 1 Sequential code

Input:
A: the coefficient matrix n x n
b: the right hand side vector n x 1
N: the number of iterations
x0: the vector x at the previous iteration
Output: **x1**

```
for (int k=0; k<N; k++) do                                ▷ iterations
    for (int i=0; i<n; i++) do                            ▷ components of x1
        float sum=0;
        for (int j=0; j<i; j++) do
            sum+=A[i][j] * x0[j];
        end for                                           ▷ split indexes
        for (int j=i+1; j<n; j++) do
            sum+=A[i][j]*x0[j];
        end for
        x1[i]=(b[i]-sum)/A[i][i];                         ▷ new x1
    end for
    x0=x1;                                                ▷ update x0
end for
```

3 Parallelization

In order to parallelize the algorithm we had to analyse its sequential version. In particular we noticed that the `for` loop on the single horizontal components of `x1` could have been done in parallel since the only writing in memory involves a different element (`x1[i]`)in each computation, to vary of `i`. So, the best strategy was a *data parallel* approach in which we partitioned the data in blocks, according to the parallel degree.

`C++` threads

For the threads implementation we fixed at run time the number of threads (`nthread`) and according to that we decide the dimension of each "block" of variable which will be assigned to each thread. We decided to use a balanced distribution of the data so each thread would do almost the same amount of work. This quantity depends surely also on the size of the input matrix A, calling it `n`. In an ideal situation the chunk dimension is:

$$\texttt{mean} = \frac{\texttt{n}}{\texttt{nthread}},$$

but in a real situation, i.e when `n` it's not divisible by `nthread`, we decided to use the approximation:

$$\texttt{mean} = \left\lfloor \frac{\texttt{n}}{\texttt{nthread}} \right\rfloor + 1.$$

So in an average situation each thread works with more variables, except for the "last thread" which computes exactly $\texttt{n} - (\texttt{mean} \times (\texttt{nthread} - 1))$ of them. In the worst case this last thread remains unused, but these situations are very infrequent and happen usually when `n` is small and clearly when $\texttt{nthread} \approx \texttt{n}$.

Barrier

The first version of the code provided a simple parallelization process in which each thread executed the `for` loops (Lines 2-11 of 1) on its block of variables and the main loop. So for each iteration `mean` threads were created and at the end of it the were all joined. This implied that we payed the cost of accessing and deleting every single thread every time.

To solve this problem we modified the structure of the body function to be passed to each thread: in this last version each thread remains active until the end of the

iterations. In fact we noticed that each thread, in each iteration, affects the same block of variables and so it's not needed to release it until the end of the iterations. To do that we used a `std::Barrier` object properly set to manage the operations end of the iterations.
This optimization increased the speedup of this version so much that we almost reached `nthread`, with an efficiency of 0.8 on average.
In the next Section 4 you will see another optimization on the stopping criterion which produced an efficiency of 0.92 on average.

Below are reported the codes for the body function of each thread and the function for the Barrier object, Algorithm 2 and Algorithm 3 respectively.

Algorithm 2 Body function of threads

Implicit Input (by reference):
A: the coefficient matrix n x n
b: the right hand side vector n x 1
NrIter: the number of iterations
x0: the vector x at the previous iteration
Explicit Input:
m: index of the first variable to compute
M: index of the first variable of the next chunk
Output: **x1[m],...,x1[M-1]**

```
1: while (NrIter>0) do
2:     for (int h=m; h<M; h++) do
3:         Internal loops as sequential version
4:         Update of x1[i]
5:     end for
6:     Barr.arrive_and_wait();
7: end while
```

Algorithm 3 Barrier object function (Barr)

Implicit Input (by reference): **NrIter**, **x0**, **x1**, **nthread**

```
1: NrIter=NrIter-1;
2: x0=x1;
```

Fast Flow

Another way to parallelize the sequential algorithm is using the Fast Flow library. This algorithm has almost the same code as the sequential one, but instead of a simple `for` loop on each thread we called:

```
pf.parallel_for(0, n, 1, 0, body)
```

Where pf was an object of type `ParallelFor` constructed with `nthread` as input and `body` is a function constructed with the internal cycle of the sequential code (Lines $3-11$). The other parameters are used for the definition of the interval of value over which we have to loop on $(0, n)$ and for the size of the steps and the dimension of the various chunks.

4 Stopping condition

The last important observation regards the iterations to be performed. Consulting the results we noticed that just the first relatively few iterations produced a real change in the vector `x1` with refer to the previous `x0`, the others, from a certain point, produced very very small variations. To avoid useless operations we decided to fix a threshold on this variations (calling it ε) so that when after an iteration the difference between the previous `x0` and the actual `x1` is less or equal than ε we could interrupt the iterations and produce the final result.

This criterion had to be added immediately before the end of each iteration, so in case of Sequential and Fast Flow version we just needed to insert the condition right before the end of the most external `for` loop (Line 12 of Alg 1):

```
if (difference(x0,x1)) then
    break the main loop;
else
    x0=x1;
end if
```

where the function `difference(x0,x1)` returns `true` iff $\frac{\|x0-x1\|_1}{n} \leq \varepsilon$, `false` otherwise.

In case of `C++` threads we tried to to the same, but to obtain a coherent result we needed to put those lines of code into the Barrier function, because each thread body function loops on all the iterations. After these modifications the value of the speedup increased, but not as much as wanted, so we tried to parallelized also this computation. Indeed, the function `difference(x0,x1)` presented before needs to do a `for` loop on each variable to compute the norm 1[1] of the difference between the two vector. So, we divided the loop among the threads and in each one we computed the norm 1 of the difference between the vector `x0` and vector `x1` for its block of variables. Then, in the Barrier object function, we made the sum of the various norms just computed[2] and we divided this result by the number of variables (`n`). The check (function `difference` from Alg 4) was then between numbers and so very cheap to be done.

With this optimization, as already said, we reach high values of speedup and efficiency that will be analysed in the next Section 5.

Algorithm 4 Barrier object function (Barr) with stopping conditions

Implicit Input (by reference):
NrIter: number of iterations
x0: the vector x at the previous iteration
x1: the vector x at the actual iteration
norm: vector `nthread` x 1 of the partial norms

```
 1: for (int y=0; y<nthread; y++) do
 2:     Norm+=norm[y];
 3:     norm[y]=0;
 4: end for
 5: Norm=Norm/((float)(n));
 6: if difference(Norm)) then                ▷ difference($f$) returns true iff $f \leq \varepsilon$
 7:     NrIter=0;
 8: else
 9:     NrIter=NrIter-1;
10:     x0=x1;
11:     Norm=0;
12: end if
```

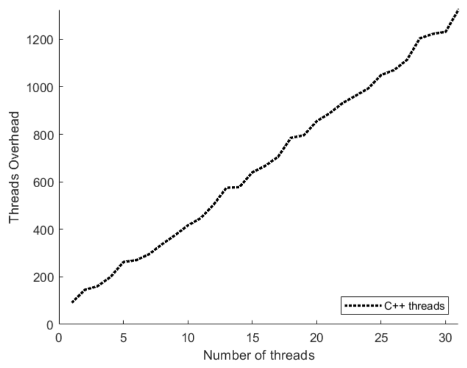
[1] Remember that the norm 1 of a vector $x \in \mathbb{R}^n$ is defined as: $\sum_{i=1}^{n} |x_1|$

[2] In the case of norm 1 we used that the sum of norms is equal to the norm of the sum, but in general this property is false for a generic norm.

5 Costs analysis

This last section will be about the various tests we executed in order to compare the proposed approaches. In particular we generated two different matrices A, one of size 10000 and the other of size 1000, with the methods of Section 1 and two corresponding vectors b. The tests were performed executing the program 5 times for each threads number, from 1 to 32, and taking from each execution the:
- Sequential time: `Tseq`
- Threads time: `Tpar(n)`
- Fast Flow time: `Tff(n)`
- Overhead for `threads`: `Tover(n)`

using the provided class `utimer`.
Then we computed the classic means between the results and obtained the first 4 columns of Tabs 1 and 2. The sequential times were not included there, but for exhaustiveness we reported them here: $\texttt{Tseq}_{10000} = 3816390\mu$ and $\texttt{Tseq}_{1000} = 39308\mu$. From this data we compute the average *SpeedUp*, *Efficiency* and *Scalability*[3] for both the threads and the Fast Flow approach. All the results are reported in the Table 1 and 2. Our idea was to exploit as many CPUs as possible and with this simple comparison we proved that to do that we need to increase the size of the system. In fact what we can simply see in the plots 1 in case of size 10000 and 1000 is that the overhead remains almost the same in both the computations, but clearly, as you can observe on the Cost Tables 1 and 2, the service times are very different from each other. This phenomenon was predictable and clearly implies that increasing the number of threads, after a certain (low) threshold reduces the efficiency of the program. Obviously the 10000 size system has the same behaviour, but after a higher number of threads. You can observe all of that in Fig 3.
Also, to evaluate the difference between the Fast Flow and Threads approach was useful to study the two systems since on the one of size 10000 their performances were almost the same, and so difficult to compare, while on the other the Fast Flow approach dominates the speedup (Fig 2). In the end, the Scalability (Fig 4) is almost optimal (compared with the speedup), but, as already said, better in the bigger system. For all the experiments the variable for the stopping criterion was $\varepsilon = 10^{-11}$, and the number of iterations produced was usually in the order of $20 - 30$ respectively for the matrix of order 10000 and 1000.



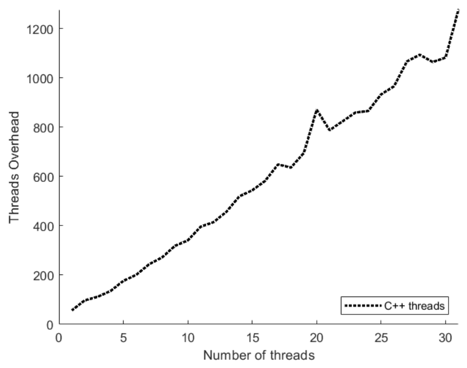


Figure 1: `C++` overhead for matrices of size $n = 10000$ and $n = 1000$

[3]$\textbf{SpUp} = \frac{\texttt{Tseq}}{\texttt{Tpar(n)}}$, $\textbf{Eff} = \frac{\texttt{Tseq}}{\texttt{Tpar(n)}*\texttt{n}}$, $\textbf{Scal} = \frac{\texttt{Tpar(1)}}{\texttt{Tpar(n)}}$

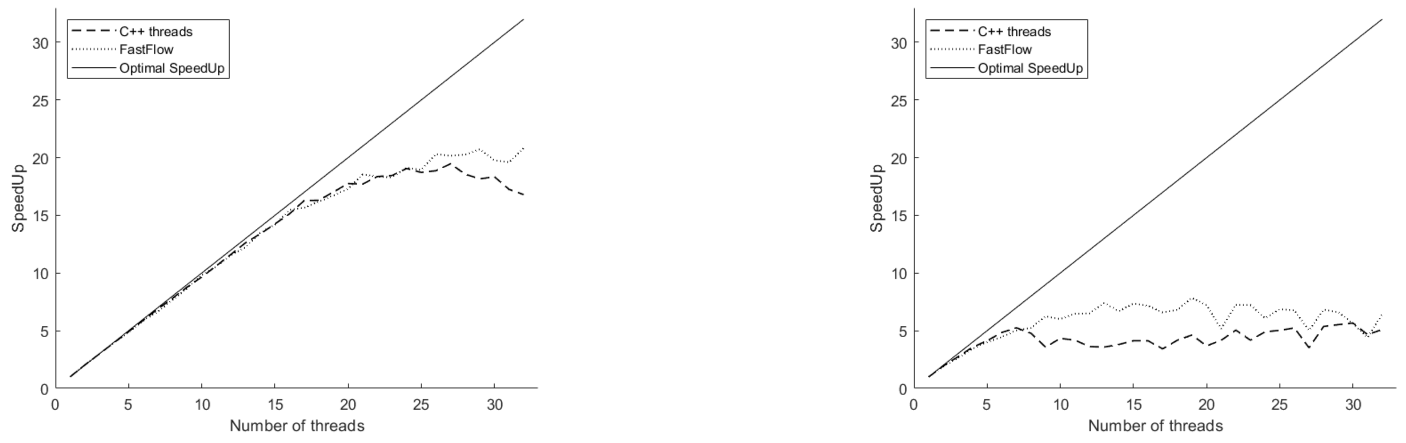


Figure 2: Speedup for matrices of size $n = 10000$ and $n = 1000$

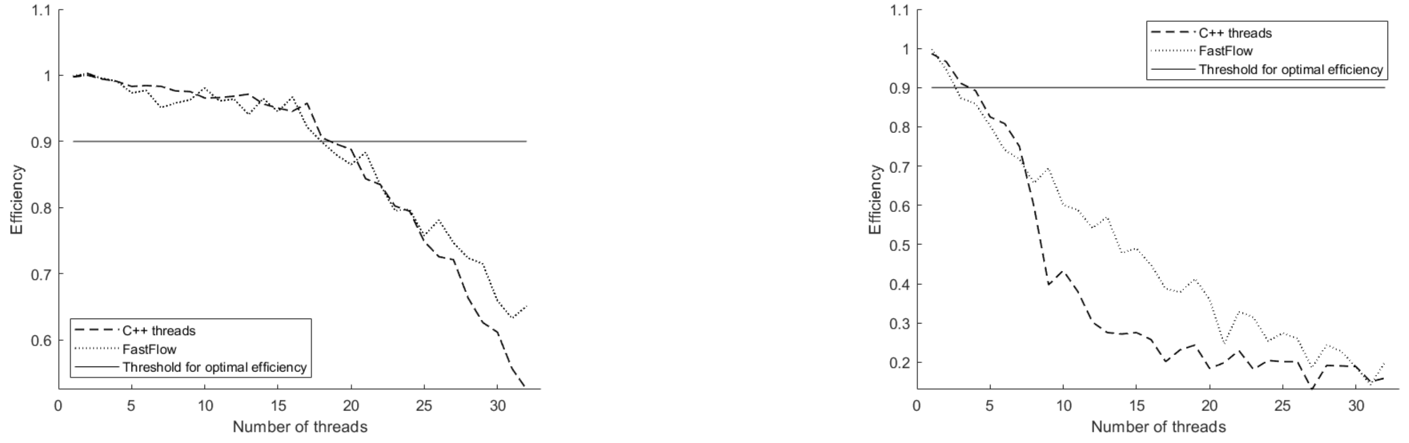


Figure 3: Efficiency for matrices of size $n = 10000$ and $n = 1000$

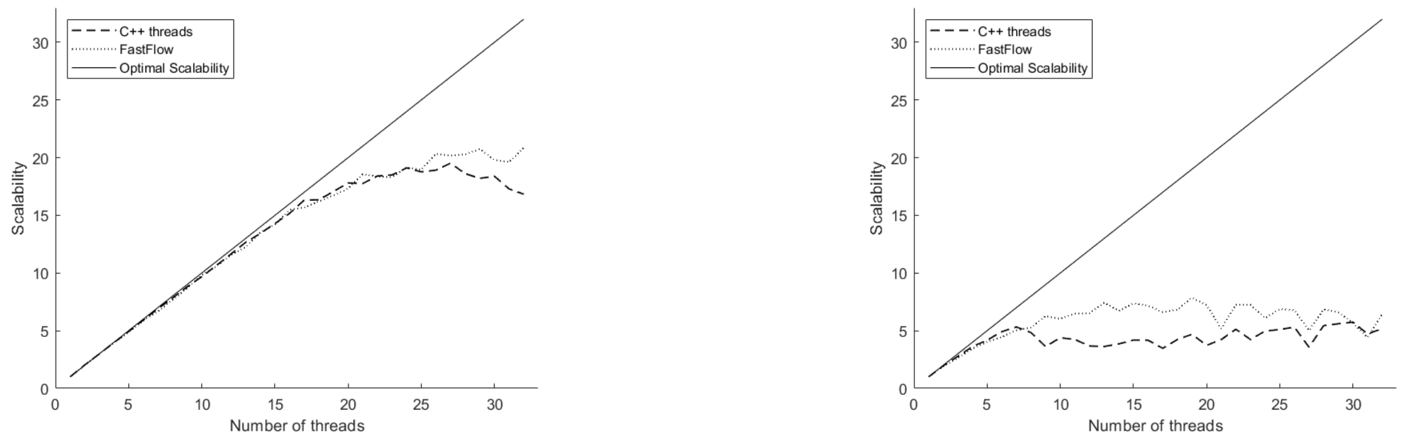


Figure 4: Scalability for matrices of size $n = 10000$ and $n = 1000$

NrT	Tover(n)	Tpar(n)	Tff(n)	SpUp par	SpUp ff	Eff par	Eff ff	Scal par	Scal ff
1	91000	38260610	38198390	0.9975	0.9991	0.9975	0.9975	1.0000	1.0000
2	145000	18891744	19025970	2.0000	1.9970	1.0000	1.0000	1.9253	1.9977
3	161000	12794424	12780772	2.9829	2.9860	0.9943	0.9943	2.9904	2.9887
4	199400	9629084	9627528	3.9634	3.9640	0.9908	0.9908	3.9734	3.9676
5	262200	7763566	7841852	4.9158	4.8667	0.9832	0.9832	4.9282	4.8711
6	270200	6461308	6509494	5.9065	5.8628	0.9844	0.9844	5.9215	5.8681
7	295400	554457	5731770	6.8831	6.6583	0.9833	0.9833	6.9006	6.6643
8	337400	4886218	4979700	7.8105	7.6639	0.9763	0.9763	7.8303	7.6708
9	375000	4347462	4403468	8.7784	8.6668	0.9754	0.9754	8.8007	8.6746
10	416400	3952198	3891594	9.6564	9.8068	0.9656	0.9656	9.6808	9.8156
11	447000	3590496	3609242	10.6291	10.5739	0.9663	0.9663	10.6561	10.5835
12	503800	328417	3299152	11.6206	11.5678	0.9684	0.9684	11.6500	11.5782
13	575200	3022052	3120700	12.6285	12.2293	0.9714	0.9714	12.6605	12.2403
14	577600	284934	2825320	13.3939	13.5078	0.9567	0.9567	13.4279	13.5200
15	639600	2677198	2692238	14.2552	14.1755	0.9503	0.9503	14.2913	14.1883
16	667000	2522330	2465940	15.1304	15.4764	0.9457	0.9457	15.1688	15.4904
17	705200	2344174	2435598	16.2803	15.6692	0.9577	0.9577	16.3216	15.6834
18	785200	2340738	2357874	16.3042	16.1857	0.9058	0.9058	16.3455	16.2004
19	796000	2241818	2283444	17.0236	16.7133	0.8960	0.8960	17.0668	16.7284
20	855000	2149086	2205600	17.7582	17.3032	0.8879	0.8879	17.8032	17.3188
21	888400	2154658	2057294	17.7123	18.5505	0.8434	0.8434	17.7572	18.5673
22	931200	2078292	2080198	18.3631	18.3463	0.8347	0.8347	18.4096	18.3629
23	961800	2068156	2086234	18.4531	18.2932	0.8023	0.8023	18.4999	18.3097
24	993200	2001548	1994736	19.0672	19.1323	0.7945	0.7945	19.1155	19.1496
25	1049600	2038822	2014882	18.7186	18.9410	0.7487	0.7487	18.7660	18.9581
26	1070400	2022688	1879228	18.8679	20.3083	0.7257	0.7257	18.9157	20.3266
27	1113200	1960196	1892986	19.4694	20.1607	0.7211	0.7211	19.5188	20.1789
28	1203800	2055190	1883372	18.5695	20.2636	0.6632	0.6632	18.6166	20.2819
29	1222600	2101512	1840856	18.1602	20.7316	0.6262	0.6262	18.2062	20.7503
30	1231200	2080126	1929608	18.3469	19.7781	0.6116	0.6116	18.3934	19.7959
31	1324600	2212082	1946650	17.2525	19.6049	0.5565	0.5565	17.2962	19.6226
32	1324610	2271974	1831970	16.7977	20.8322	0.5249	0.5249	16.8402	20.8510

Table 1: Costs table for a 10000 order matrix (μ)

NrT	Tover(n)	Tpar(n)	Tff(n)	SpUp par	SpUp ff	Eff par	Eff ff	Scal par	Scal ff
1	56600	39845	39408	0.9865	0.9975	0.9865	0.9975	1.0000	1.0000
2	97400	20340	20778	1.9325	1.8918	0.9663	0.9459	1.9589	1.8966
3	111800	14379	15003	2.7336	2.6200	0.9112	0.8733	2.7710	2.6267
4	135200	11012	11438	3.5696	3.4367	0.8924	0.8592	3.6184	3.4455
5	176400	9526	9793	4.1264	4.0137	0.8253	0.8027	4.1828	4.0239
6	201000	8108	8840	4.8481	4.4467	0.8080	0.7411	4.9143	4.4580
7	244000	7482	7827	5.2537	5.0221	0.7505	0.7174	5.3254	5.0349
8	271600	8234	7479	4.7739	5.2559	0.5967	0.6570	4.8391	5.2693
9	318400	10990	6290	3.5767	6.2495	0.3974	0.6944	3.6256	6.2654
10	340800	9062	6541	4.3376	6.0093	0.4338	0.6009	4.3968	6.0246
11	396400	9404	6078	4.1800	6.4675	0.3800	0.5880	4.2371	6.4839
12	414800	10840	6041	3.6261	6.5064	0.3022	0.5422	3.6756	6.5230
13	457000	10973	5303	3.5823	7.4124	0.2756	0.5702	3.6313	7.4313
14	519400	10329	5864	3.8057	6.7033	0.2718	0.4788	3.8577	6.7203
15	544000	9514	5346	4.1316	7.3528	0.2754	0.4902	4.1880	7.3715
16	582200	9531	5486	4.1241	7.1651	0.2578	0.4478	4.1805	7.1834
17	648800	11472	5968	3.4265	6.5867	0.2016	0.3875	3.4733	6.6034
18	636000	9418	5770	4.1739	6.8125	0.2319	0.3785	4.2309	6.8298
19	695400	8495	5011	4.6272	7.8443	0.2435	0.4129	4.6904	7.8643
20	870600	10671	5458	3.6837	7.2016	0.1842	0.3601	3.7340	7.2200
21	787800	9421	7597	4.1724	5.1743	0.1987	0.2464	4.2294	5.1874
22	823600	7776	5430	5.0553	7.2385	0.2298	0.3290	5.1244	7.2569
23	859200	9404	5435	4.1801	7.2319	0.1817	0.3144	4.2372	7.2502
24	866000	8012	6455	4.9061	6.0895	0.2044	0.2537	4.9732	6.1050
25	933400	7810	5726	5.0332	6.8651	0.2013	0.2746	5.1019	6.8825
26	966000	7498	5810	5.2422	6.7656	0.2016	0.2602	5.3138	6.7828
27	1066800	11154	7839	3.5240	5.0147	0.1305	0.1857	3.5721	5.0274
28	1094000	7324	5756	5.3673	6.8293	0.1917	0.2439	5.4406	6.8467
29	1064400	7114	5952	5.5258	6.6037	0.1905	0.2277	5.6012	6.6205
30	1082400	6934	6965	5.6687	5.6436	0.1890	0.1881	5.7462	5.6580
31	1276800	8433	8907	4.6610	4.4133	0.1504	0.1424	4.7247	4.4245
32	1276810	7699	6112	5.1055	6.4309	0.1595	0.2010	5.1752	6.4472

Table 2: Costs table for a 1000 order matrix (μ)