Project Proposal PHPC-2017.

Gauss-Seidel Method with Successive Over-relaxation For Solving System of Linear Equations

Principal Investigator	Olagoke Lukman Olabisi
Institution	EPFL-SCITAS
Address	Station 1, CH-1015 LAUSANNE
Involved Researcher	Only PI
Date of submission	May 14 2017
Expected End of Project	June 30 2017
Target Machine	Bellatrix
Proposed Pseudonym	GS or GS-SOR

1 Introduction

The Gauss Siedel method is an iterative method that can be used for linear systems of equations

$$Ax = b$$

It is usually twice as fast as the Jacobi algorithm but not as easily parallelizable.

2 The Gauss Siedel Mathod

Let's define: $A\mathbf{x} = \mathbf{b}$
matrix equation for a system of linear equations where A the m x n matrix, $\mathbf{x}$ is a column vector with n entries, and $\mathbf{b}$ is a column vector with m entries. Then the Gauss-Siedel method [1] is defined by the iteration:

$$L_*\mathbf{x}^{(k+1)} = \mathbf{b} - U\mathbf{x}^{(k)}$$

where $\mathbf{x}^{(k)}$ is the kth approximation or iteration of x , $x^{(k+1)}$ is the next or $k+1$ iteration of $\mathbf{x}$, and the matrix A is decomposed into a *lower triangular component* L_*, and a *strictly upper triangular component* $\mathbf{U}$: That is

$$A = L_* + U$$

.

where :

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{n4} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

And:

$$L_* = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \mathbf{U} = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix}$$

The system of linear equation can rewritten as: The system of linear equations may be rewritten as:

$$L_*\mathbf{x} = \mathbf{b} - U\mathbf{x}$$

The GaussSeidel method now solves the left hand side of this expression for x, using previous value for x on the right hand side. Analytically, this may be written as:

$$\mathbf{x}^{(k+1)} = L_*^{-1}(\mathbf{b} - U\mathbf{x}^{(k)})$$

However, by taking advantage of the triangular form of L_* , the elements of $x(k+1)$ can be computed sequentially using forward substitution:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n. \tag{1}$$

The convergence of GS can effectively be accelerated if we introduce a relaxation ω so that (1) becomes:

$$x_i^{(k+1)} = (1-\omega) x_i^{k-1} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n. \tag{1}$$

The matrix notatation (1) can also be composed as : $(D - L)x^{k+1} = Ux^k + b, k \geq 0$ where L, D, and U are the lower, diagonal, and upper-triangular parts of matrix A respectively.

3 Sequential GS Implementation

To evaluate the performance of GS algorithm one must first look at the sequential performance. The sequential program was ran on the Bellatrix cluster at EPFL. This machine presents the following characteristics:

- 424 compute nodes each with 2 Intel 8-cores Sandy-Bridge CPU running at 2.2 GHz, with 32 GB of memory
- Infiniband QDR 2:1 connectivity
- GPFS filesystem

However, since this aim is to first evaluate the performance of the sequential execution, there was no need to exploit full capacity of the computation nodes the clustor has to offer. However, compiler optimization was carried out to boost performance. The sequential implementation code in C++. We also investigated the sequential GS-SOR to increase the convergence of the algorithm.

3.1 Performance Analysis for Sequential Implementation

It is well known that convergence is guaranteed only if matrix A is diagonally dominant or symmetric and positive semi-definite. Matrix input of various sizes were investigated $n = (32, 64, 128, ..)$.

Figure 1 shows the relationship between input Array size and the execution time (in seconds) needed for the solution of the problem. From the figure we observe that the convergence of GS algorithm is dependent on the input Array size and the number of iterations increases as the size becomes large (it is of order N^2, O(N^2),and for k iterations we need k(N^2) operations. Hence the sequential implementation might not scale well on large problem size typical in real world scientific applications.

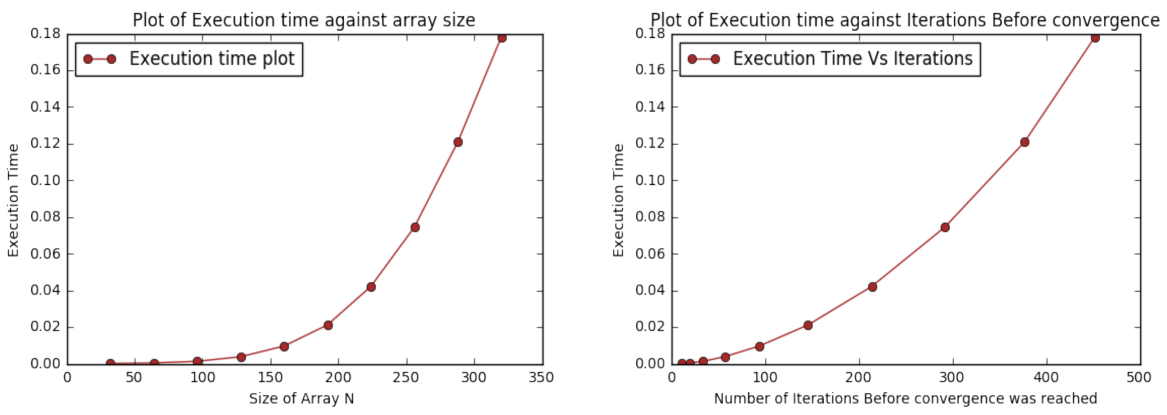
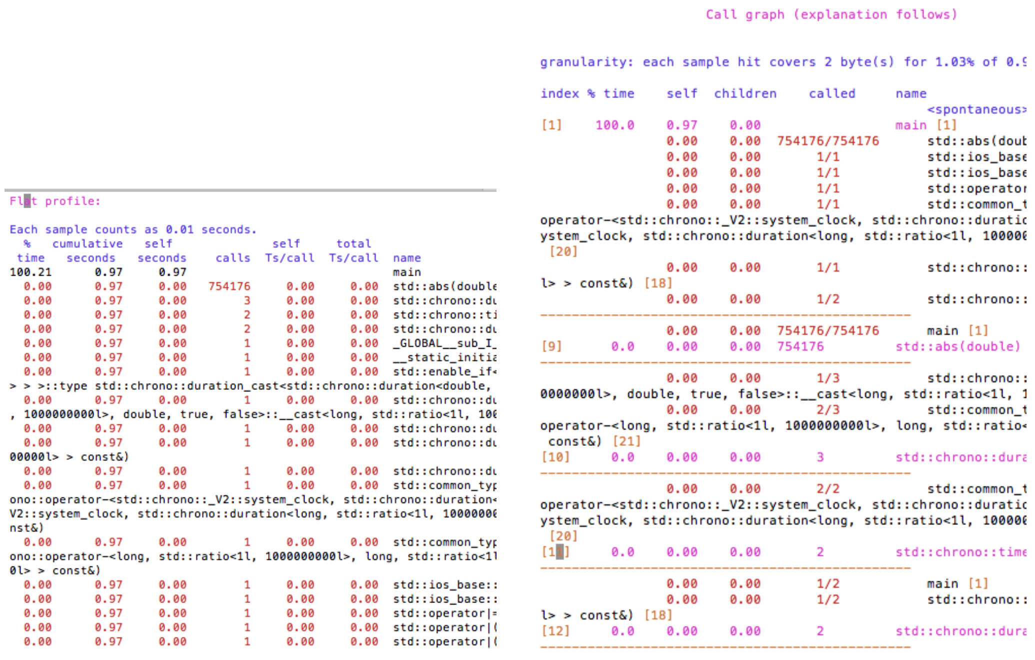


Figure 1: Caption

4 Flat profiling

The profile was generated using an array size of n = 512. As we can observe most of the time was spent in the main function since it was the only function called. It is observed that the more the stopping criteria value is increased the time to completion is also increased. With this we can see that convergence will be a problem to watch out for in the parallel implementation. And further scrutiny shows that most time is spent in the loop trying to in the loop where we specify the vales should be less than certain values (convergence value) before the loop ends.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.21      0.97     0.97                             main
  0.00      0.97     0.00   754176     0.00     0.00  std::abs(double
  0.00      0.97     0.00        3     0.00     0.00  std::chrono::du
  0.00      0.97     0.00        2     0.00     0.00  std::chrono::ti
  0.00      0.97     0.00        2     0.00     0.00  std::chrono::du
  0.00      0.97     0.00        1     0.00     0.00  _GLOBAL__sub_I_
  0.00      0.97     0.00        1     0.00     0.00  __static_initia
  0.00      0.97     0.00        1     0.00     0.00  std::enable_if<
> > >::type std::chrono::duration_cast<std::chrono::duration<double,
  0.00      0.97     0.00        1     0.00     0.00  std::chrono::du
, 1000000000l>, double, true, false>::__cast<long, std::ratio<1l, 100
  0.00      0.97     0.00        1     0.00     0.00  std::chrono::du
  0.00      0.97     0.00        1     0.00     0.00  std::chrono::du
00000l> > const&)
  0.00      0.97     0.00        1     0.00     0.00  std::chrono::du
  0.00      0.97     0.00        1     0.00     0.00  std::common_typ
ono::operator-<std::chrono::_V2::system_clock, std::chrono::duration<
V2::system_clock, std::chrono::duration<long, std::ratio<1l, 1000000
nst&)
  0.00      0.97     0.00        1     0.00     0.00  std::common_typ
ono::operator-<long, std::ratio<1l, 1000000000l>, long, std::ratio<1l
0l> > const&)
  0.00      0.97     0.00        1     0.00     0.00  std::ios_base::
  0.00      0.97     0.00        1     0.00     0.00  std::ios_base::
  0.00      0.97     0.00        1     0.00     0.00  std::operator|=
  0.00      0.97     0.00        1     0.00     0.00  std::operator|(
  0.00      0.97     0.00        1     0.00     0.00  std::operator|(
```

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 1.03% of 0.9

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.97    0.00                 main [1]
                0.00    0.00  754176/754176      std::abs(doub
                0.00    0.00       1/1           std::ios_base
                0.00    0.00       1/1           std::ios_base
                0.00    0.00       1/1           std::operator
                0.00    0.00       1/1           std::common_t
operator-<std::chrono::_V2::system_clock, std::chrono::duratio
ystem_clock, std::chrono::duration<long, std::ratio<1l, 100000
 [20]
                0.00    0.00       1/1           std::chrono::
l> > const&) [18]
                0.00    0.00       1/2           std::chrono::
-----------------------------------------------
                0.00    0.00  754176/754176      main [1]
[9]      0.0    0.00    0.00  754176         std::abs(double)
-----------------------------------------------
                0.00    0.00       1/3           std::chrono::
0000000l>, double, true, false>::__cast<long, std::ratio<1l, 1
                0.00    0.00       2/3           std::common_t
operator-<long, std::ratio<1l, 1000000000l>, long, std::ratio<
 const&) [21]
[10]     0.0    0.00    0.00       3         std::chrono::dura
-----------------------------------------------
                0.00    0.00       2/2           std::common_t
operator-<std::chrono::_V2::system_clock, std::chrono::duratio
ystem_clock, std::chrono::duration<long, std::ratio<1l, 100000
 [20]
[11]     0.0    0.00    0.00       2         std::chrono::time
-----------------------------------------------
                0.00    0.00       1/2           main [1]
                0.00    0.00       1/2           std::chrono::
l> > const&) [18]
[12]     0.0    0.00    0.00       2         std::chrono::dura
-----------------------------------------------
```

Figure 2: Flat profile and call graph with convergence value = 1e-6 and Array size N = 512

The effect of compiler optimization was investigated and the outcome is presented in the table below:

optimization level	g++	Number of Iterations Until Convergence
Not assigned	0.9702	961
O	0.7712	960
O2	0.2688	962
O3	0.2665	961
O1 and ftree-vectorizer	0.7504	961
O2 and ftree-vectorizer	0.2617	961
O3 and ftree-vectorizer	0.2645	962

From the table above we can see our performance range is between 0.9702 and 0.2545 even after the compiler optimization and the number of iterations till all vallues converges within the set tolerance range is same. In the parallel execution a relaxation parameter will be introduced to aid convergence - in addition to the tolerance value for convergence.

5 Proposed Parallel Implementation

Since we observe that the sequential execution does not scale for large problem size. This fact forms is the basic motivation for a scalable parallel execution.

The proposed parallel algorithm uses operate as follows

- An array of size q by N is instantiated, where q is the number of groups and N is the problem size.
- The groups will be laid out in such a way as to reduce communication between groups .
- Each group will be structured to partition its nodes into interior nodes and boundary nodes.
- The interior nodes can operate on GS iteration loop without communication.
- Only boundary nodes requiring massage passing for the updates values x^{s+1} at each step(s) of the GS iteration. The simple architectural view is shown below: Implemen-

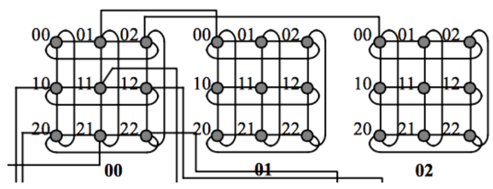


Figure 3: Communication structure for parallel implementation

tation wise this could be reduced to the following

1. For i = 2 to q do :
2. Group i sends x^{s+1} back to group $i - 1$,
3. receives x^{s} from group $i + 1$,
4. receives x^{s+1} from group $i{-}1$

5. Call GS subroutine
6. Send x^{s+1} to group i + 1
7. Continue Till convergence

The actual implementation will use some realxation parameters to aid convergence.

6 Conclusion

When the same implement The element-wise formula for the GaussSeidel method is quite tricky to parallelize. Nevertheless this work has laid the motivation or need for a parallel implementation. One subtle thing to be noted is the need for convergence. This should be adequately taken care of in the parallel execution otherwise the algorithm will never converge and we will never be able to make further analysis.

In addition a roof-line model analysis of the parallel execution will be presented. Furthermore the performance of the parallel execution with number of nodes will be analyzed. The speed up and parallel efficiency using Amdah's law will be investigated too.

References

[1] Gaussian-seidel method, wikipedia.