EFFICIENT PARALLELIZATION OF LANCZOS TYPE ALGORITHMS

ILYA POPOVYAN
MOSCOW STATE UNIVERSITY, RUSSIA

Abstract. We propose a new parallelization technique for Lanzos type algorithms for solving sparse linear systems over finite fields on mesh cluster architecture. The algorithm computation time scales as P^{-1} on P processors, and the communication time scales as $P^{-1/2}$ for reasonable choice of P.

1. Introduction

Most of the stages of the general number field algorithm [1], the best known algorithm for factoring and discrete logarithms at the moment, are known to be easily parallelized or even distributed between almost independent computation units. But it is not exactly the case for the stage when the large sparse linear system has to be solved.

Currently there are two major approaches to this problem: iterative algorithms of Wiedeman [2] and Lanczos [3] type . The former is a two-stage algrithm and is very popular among the specialists [4] due to its native distributability on the first stage. The weakness of the Wiedeman type algorithms is the second stage when the solution of the linear system should be constructed by redoing all the matrix-vector multiplications.

Lanczos type algorithms on the other hand are known to be hard to parallelize. But in the same time they don't have any second stage and yield a solution without extra matrix-vector multiplications.

There are few publications on how to parallelize Lanczos type algorthims [5], [6], [7], [8]. All of them employ different matrix partitioning strategies and cluster architectures. In this article we briefly describe the ideas behind these algortihms, present our algorithm and compare their running times in terms of number of processors. We notice that the parallelization techniques we describe are applicable with minor changes to all Lanczos type algorithms over finite fields including randomized Lanczos algorithms over $GF(p)$ [9] and Montgomery version [10] of block Lanczos over $GF(2)$.

2. Lanczos algorithms

Let p be a prime. There is a linear system

$$Cx = y, C \in GF(p)^{m\times n}, x \in GF(p)^n, y \in GF(p)^m, \tag{2.1}$$

with large $n, m \in \mathbb{N}$. We call the matrix C *sparse* if there exists $d \in \mathbb{N}, d \ll n$ such that the number of nonzero coefficients in each row of C is less than d.

Date: August 2, 2011.

Suppose C is not symmetric, premultiply the system by C^t and change the variables

$$A = C^tC \in GF(p)^{n\times n}, b = C^ty \in GF(p)^n. \tag{2.2}$$

If $y = 0$ let us select a random $0 \neq z \in GF(p)^n$ and set $y = Cz$. Now try to solve a symmetric system (2.2) with nonzero b on the rigth side. When the solution of (2.2) x is obtained we hope to be able to construct the solution x' of (2.1) since $Ax - b = C^t(Cx - y) = 0$. If y was zero we then obtain a solution of (2.1) as $x' - z$ since $Cx' - y = C(x' - z) = 0$.

The core of the Lanczos type algorithms for symmetric A is the following iteration:

$$\begin{aligned} &w_0 = b, w_1 = Aw_0 - \tfrac{w_0^tA^2w_0}{w_0^tAw_0}w_0, \\ &w_{i+1} = Aw_i - \tfrac{w_i^tA^2w_i}{w_i^tAw_i}w_i - \tfrac{w_i^tA^2w_{i-1}}{w_{i-1}^tAw_{i-1}}w_{i-1}, i = 1, \ldots, n-1. \end{aligned} \tag{2.3}$$

The iterations stop when $w_j : w_j^tAw_j = 0$. If the last vector $w_j = 0$, one could construct the solution using the formula

$$x = \sum_{i=0}^{j-1} \frac{b^tw_i}{w_i^tAw_i}w_i.$$

Block Lanczos type algorithms are slightly different as they use subspaces instead of vectors w_i and matrices instead of scalars in (2.3). But essensially the form of the iteration stays the same.

2.1. **Implementation remarks.** A good implementation of the Lanczos type algorithm should exploit its iterative nature in order to get rid of unnessesary time consuming operations. For instance, one should remember $w_{i-1}^tAw_{i-1}$ from the previous iteration and not recalculate it; or one could use the recurrent formula

$$b^tw_{i+1} = -\frac{w_i^tA^2w_i}{w_i^tAw_i}(b^tw_i) - \frac{w_i^tA^2w_{i-1}}{w_{i-1}^tAw_{i-1}}(b^tw_{i-1})$$

to avoid direct vector-vector multiplications.

But the two most important and time consuming operations on each iteration – sparse matrix-vector (*matvec*) multiplication Aw_i and two vector-vector (*vecvec*) multiplications $w_i^tAw_i, w_i^tA^2w_i$ – will always be present in any implementation.

If C is not symmetric it is crucial that the multiplication by A should be done by first multiplicating by sparse matrix C and then by C^t. This (double) operation is the most time consuming and is the first candidate to be parallelized.

One also could notice that in general case $w_i^tAw_i, w_i^tA^2w_i$ are 'scalar squares', i.e. $w_i^tAw_i = (Cw_i)^t(Cw_i), w_i^tA^2w_i = (Aw_i)^t(Aw_i)$. This means that matvec Aw_i and two vecvec $w_i^tAw_i, w_i^tA^2w_i$ could be seen as one matvec Cw_i and one vecvec $(Cw_i)^t(Cw_i)$ followed by the transposed operation – one matvec $C^t(Cw_i)$ and one vecvec $(C^tCw_i)^t(C^tCw_i)$.

3. Parallelization techniques

There are few approaches to the parallel implementation of Lanczos type algorithms [5], [6], [7], [8]. The main difference between them is in the matrix distribution on the processors and the cluster architectures. All of the approaches aim to effectively parallelize the matvec, some of them also parallelize the vecvecs and other vector operations.

Operation	Data before	Data after
a. Broadcast	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1 & \cdots & \end{array}$	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1 & \cdots & x_1 \end{array}$
b. Reduce (+)	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1 & \cdots & x_P \end{array}$	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline \sum_{i=1}^{P} x_i & \cdots & \end{array}$
c. Scatter	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1, \ldots, x_P & \cdots & \end{array}$	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1 & \cdots & x_P \end{array}$
d. Gather	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1 & \cdots & x_P \end{array}$	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1, \ldots, x_P & \cdots & \end{array}$
e. Allgather	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1 & \cdots & x_P \end{array}$	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1, \ldots, x_P & \cdots & x_1, \ldots, x_P \end{array}$
f. Allreduce (+)	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_1 & \cdots & x_P \end{array}$	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline \sum_{i=1}^{P} x_i & \cdots & \sum_{i=1}^{P} x_i \end{array}$
g. Reduce-Scatter	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline x_{1,1} \ldots x_{1,P} & \cdots & x_{P,1} \ldots x_{P,P} \end{array}$	$\begin{array}{c\|c\|c} \mathbf{1} & \cdots & \mathbf{P} \\ \hline \sum_{i=1}^{P} x_{i,1} & \cdots & \sum_{i=1}^{P} x_{i,P} \end{array}$

TABLE 1. Global operations

3.1. **Global operations.** The major problem in the parallel matvec is a need in *global (collective) operations*, the communication/calculation operations which involve a group of processors.

There are four basic global operations in MPI [11]. These are: broadcast, reduce, scatter and gather. There are also compound global operations which could be implemented using the basic ones, among them allreduce, allgather, reduce-scatter. Table 1 shows what each of them does.

Native MPI implementation of these operations uses a tree of processors, this gives a factor of $\log P$ in the communications cost. But for a bigger data sizes there are other implementations [12] of the global operations which run in time depending only on the data size (linearly) and not the number of processors P. We assume one of them, a bucketing algorithm, is used for global operations in the next sections.

3.2. **Parallel version 1: per element matrix distribution.** This approach is described in [5], [6].

Each processor has a subset of nonzero elements of the matrix. The distribution of elements between the processors is ballanced so each processor has approximately the same number of matrix elements.

All the algorithm computations are performed by one master processor with full length vectors, all the other processors are just helpers for the matvec operations. The matvec is performed in three stages (table 2):

a. Initial data placement	**1**	**2**	$\cdots$	**P**
	$(c_{i,j})_{I_1\times J_1}; v$	$(c_{i,j})_{I_2\times J_2}$	$\cdots$	$(c_{i,j})_{I_P\times J_P}$
b. Vector sent	**1**	$\cdots$	**P**	
	$(c_{i,j})_{I_1\times J_1}; (v_j)_{J_1}$	$\cdots$	$(c_{i,j})_{I_P\times J_P}; (v_j)_{J_P}$	
c. Partial products/sums	**1**	$\cdots$	**P**	
	$(\sum_{j\in J_1} c_{i,j}v_j)_{I_1}$	$\cdots$	$(\sum_{j\in J_1} c_{i,j}v_j)_{I_P}$	
d. Received partial sums	**1**	**2**	$\cdots$	**P**
	$\sum_{k=1}^{P}(\sum_{j\in J_k} c_{i,j}v_j)$		$\cdots$	
e. Final data placement	**1**	**2**	$\cdots$	**P**
	$(c_{i,j})_{I_1\times J_1}; Cv$	$(c_{i,j})_{I_2\times J_2}$	$\cdots$	$(c_{i,j})_{I_P\times J_P}$

TABLE 2. Parallel matvec for per element matrix distribution

(1) master decides which parts of the vector each helper processor needs for the operation and sends it to him,
(2) each helper processor recieves the vector parts and computes the products of matrix elements and appropriate vector coordinates and sends the results to master,
(3) master recieves all the partial products and summs them between the processors possessing the matrix elements of the same row.

The strength of this approach is in the ballance of the calculations: one could use any sparse matrix partitioning software [13] to achieve perfect matrix ballance. This gives a scale factor P^{-1} on P processors for each matvec of the algorthim. All other calculations are done by master processor and therefore not parallelized.

On the other hand the communication time in this approach is either constant in P or scales as $P^{1/2}$ depending on the implementation and cluster architecture.

a. Initial data placement	**1**	$\cdots$	**P**
	$C_1; V_1$	$\cdots$	$C_P; V_P$
b. (sub)Matvec	**1**	$\cdots$	**P**
	C_1V_1	$\cdots$	C_PV_P
c. Reduce-Scatter (+)	**1**	$\cdots$	**P**
	$(\sum_{i=1}^{P} C_iV_i)_1$	$\cdots$	$(\sum_{i=1}^{P} C_iV_i)_P$
d. Final data placemen	**1**	$\cdots$	**P**
	$(Cv)_1$	$\cdots$	$(Cv)_P$

TABLE 3. Parallel matvec for per column matrix distribution

3.3. **Parallel version 2: per column matrix distribution.** This approach is good only when matrix C is symmetric and no second C^t matvec is required.

Split a set of the matrix columns between the processors so that each processor has approximately the same number of columns. The matrix should be preprocessed so each processor also has approximately the same number of non zero elements.

All the algorithm computations are performed by all the processors, each having its own $1/P$ part of the vector. The matvec is performed in two stages (table 3):

(1) each processor computes matvec with its column submatrix and its part of the vector,
(2) reduce-scatter operation is applied so all the processors have their own parts of the sum of the previous stage results.

Each processor does the vecvec by first vecvecing its own parts and then allreducing (summing) the results between all the processors.

Since each processor does all the calculations with only a part of the vector, the scaling factor for the vector calculations is P^{-1}. Assuming the matrix column distibution was ballanced one could see that matvec and vecvec times also scale as P^{-1}. So all the computations are perfectly parallelized.

The communication time is in the same time still a problem – the reduce-scatter operation on the second stage of the matvec takes a constant time independently of the number of processors P.

a. Initial data placement

	1	$\cdots$	**Q**
1	$C_{1,1}; V_1$	$\cdots$	$C_{1,Q}; V_Q$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Q	$C_{Q,1}; V_1$	$\cdots$	$C_{Q,Q}; V_Q$

b. (sub)Matvec

	1	$\cdots$	**Q**
1	$C_{1,1}V_1$	$\cdots$	$C_{1,Q}V_Q$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Q	$C_{Q,1}V_1$	$\cdots$	$C_{Q,Q}V_Q$

c. Allreduce (+) in rows

	1	$\cdots$	**Q**
1	$\sum_{i=1}^{Q} C_{1,i}V_i$	$\cdots$	$\sum_{i=1}^{Q} C_{1,i}V_i$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Q	$\sum_{i=1}^{Q} C_{Q,i}V_i$	$\cdots$	$\sum_{i=1}^{Q} C_{Q,i}V_i$

d. Final data placement

	1	$\cdots$	**Q**
1	$C_{1,1}; (Cv)_1$	$\cdots$	$C_{1,Q}; (Cv)_1$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Q	$C_{Q,1}; (Cv)_Q$	$\cdots$	$C_{Q,Q}; (Cv)_Q$

TABLE 4. Parallel matvec for per submatrix matrix distribution

3.4. **Parallel version 3: per submatrix matrix distribution.** This approach is described in [8] and also implemented in *msieve* [7], one of the best open source tools for factoring.

The approach requires mesh architecture on the cluster (we will assume that $P = Q^2$ and we have $Q \times Q$ mesh) and is suitable for any kind of matrix C.

Split a matrix into P submatrices of approximately the same size and scatter them between the processors according to the row-column index in the mesh. The matrix should be preprocessed so each processor also has approximately the same number of non zero elements.

All the algorithm computations are performed by all the processors, each having its own $1/Q$ part of the vector. All the processors of the same mesh column share the part of the vector and duplicate the calculations of each other. The matvec is performed in two stages (table 4):

(1) each processor computes matvec with its submatrix and its part of the vector,
(2) allreduce operation is applied in each mesh row in parallel so all the processors in the row share the sum of the results of the previous stage.

Notice that the resulting vector parts distribution is transposed (now each mesh row has the same part). This fact is used in the subsequent C^t matvec operation performed in the same manner bringing back the initial vector parts distribution.

Each processor does the vecvec operation by first vecvecing its own parts and then allreducing (summing) the results within the mesh column (row, after the C^t matvec) in parallel.

Since each processor does all the calculations with only a part of the vector, the scaling factor for the vector calculations is Q^{-1}, also the vecvecs time scales as Q^{-1}. Assuming the submatrix distibution was ballanced one could see that matvec time scales as P^{-1}.

As to the communication time, all the allreduce operations are performed with $1/Q$ parts of the vectors in parallel, so its time scales as Q^{-1}. The scalar allreduce operation time required for the vecvecs scales as $\log Q$.

All this gives the total algorithm running time scale factor $P^{-1/2}$ for the reasonable range of mesh sizes P.

4. Our parallel version

Our parallel approach employs the per submatrix distribution on the $Q \times Q$ mesh as described above.

But unlike the previous case all the algorithm computations are performed by all the processors, each having its own $1/P$ part of the vector. The matvec is performed in three stages (table 5):

(1) allgather operation is performed in each mesh column in parallel so all the processors of the columns construct the same $1/Q$ part of the vector from $1/P$ pieces
(2) each processor computes matvec with its submatrix and the $1/Q$ part of the vector built on the previous stage,
(3) reduce-scatter vector operation is applied in each mesh row in parallel so all the processors in the row has an appropriate $1/P$ part of the vector sum of the results from the previous stage.

a. Initial data placement

	1	$\cdots$	**Q**
1	$C_{1,1}; V_{1,1}$	$\cdots$	$C_{1,Q}; V_{1,Q}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Q	$C_{Q,1}; V_{Q,1}$	$\cdots$	$C_{Q,Q}; V_{Q,Q}$

b. Allgather in columns

	1	$\cdots$	**Q**
1	$C_{1,1}; V_1$	$\cdots$	$C_{1,Q}; V_Q$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Q	$C_{Q,1}; V_1$	$\cdots$	$C_{Q,Q}; V_Q$

c. (sub)Matvec

	1	$\cdots$	**Q**
1	$C_{1,1}V_1$	$\cdots$	$C_{1,Q}V_Q$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Q	$C_{Q,1}V_1$	$\cdots$	$C_{Q,Q}V_Q$

d. Reduce-Scatter (+) in rows

	1	$\cdots$	**Q**
1	$(\sum_{i=1}^{Q} C_{1,i}V_i)_1$	$\cdots$	$(\sum_{i=1}^{Q} C_{1,i}V_i)_Q$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Q	$(\sum_{i=1}^{Q} C_{Q,i}V_i)_1$	$\cdots$	$(\sum_{i=1}^{Q} C_{Q,i}V_i)_Q$

e. Final data placement

	1	$\cdots$	**Q**
1	$C_{1,1}; (Cv)_{1,1}$	$\cdots$	$C_{1,Q}; (Cv)_{Q,1}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Q	$C_{Q,1}; (Cv)_{1,Q}$	$\cdots$	$C_{Q,Q}; (Cv)_{Q,Q}$

TABLE 5. Parallel matvec in our version

The data distribution transposition is used to perform the subsequent C^t matvec in the very same manner.

After the matvec each processor does the vecvec operation by first vecvecing its own parts of the vector and then allreducing (summing) the results between all the processors.

Since each processor does all the calculations with only a part of the vector, the scaling factor for the vector calculations is P^{-1}, also the vecvecs scale as P^{-1}. Assuming the submatrix distibution was ballanced one could see that matvec scales as P^{-1}.

The communication time in our version of the algorithm is defined by three global operations: allgather, reduce-scatter and a scalar allreduce. The first two are performed with $1/Q$ parts of the vectors in parallel, so their time scales as Q^{-1}. The latter scales as $\log Q$.

The difference of our version from the version described in paragraph 3.4 is in the scaling factor of the calculations. Basically, our version works Q times faster in calculations while takes the same time for communications.

4.1. **Communication/calculation interleaving.** Our version could also employ communication/calculation interleaving.

Split each submatrix in Q column sets (subsubmatrices) in the same way a bigger $1/Q$ vector part is split into $1/P$ parts. Now each processor does matvec of the subsubmatrix and the appropriate $1/P$ vector part while sending/receiving the other $1/P$ vector part during allgather operation. The calculations should be arranged in such a way that all the subsubmatrix matvec results are accumulated. After the last $1/P$ part of the vector is received the processor performs the last subsubmatrix matvec with it. It's easy to see that the accumulated result will be exactly the result of matvec of submatrix and a a bigger $1/Q$ vector part combined by allgather.

This technique allows to combine matvec and allgather, two very time consuming operations, resulting in the max of their running times instead of their sum. Although an effective implementation of it may require extra processor memory for different matrix values layout.

4.2. **Experimental results.** We implemented our algorithm for the classic version of Lanczos algorithm over a finite field of big characteristic and tested it on MSU 'Lomonosov' cluster. We solved a system of size 369669×185401 variables over $GF(p)$ with 64 bit p. The experiments showed that the algorithm running time scales in number of processors as predicted.

Also the experiments showed that a good matrix ballance is very important. We used a simple greedy algorithm randomly permuting columns and rows of the matrix and got the running time almost two times less.

Although our experiments with communication/calculation interleaving showed almost no acceleration, we believe this can be explained by increased overhead due to extra data conversions. We expect the communication/calculation interleaving to be more efficeint in the Mongomery version of block Lanczos algorithm over $GF(2)$ where no conversion is required.

5. Conclusions

We proposed a new parallel approach for Lanczos type algorithms. This approach yields a scale factor of P^{-1} in computation and $P^{-1/2}$ in communication time for the reasonable number of processors P. This is probably the best time that can be achieved with planar matrix partitioning. Currently we investigate other matrix partitioning strategies with a better communication time scale factor.

Acknowledgements The author thanks the Research Computing Center of M.V. Lomonosov Moscow State University for the access to the supercomputer SKIF MSU 'Lomonosov'.

References

[1] A.K. Lenstra and H.W. Lenstra Jr. *The development of the number field sieve*, vol. 1554 of Lecture Notes in Mathematics. Springer-Verlag Berlin, 1993.

[2] D. H. Wiedemann. *Solving sparse linear equations over finite fields.* IEEE Transactions on Information Theory, 32(1):54–62, Jan. 1986

[3] Lanczos C. *An Iteration Method for the Solution of the Eigenvalue Problem of Linear Dierential and Integral Operators.* Journal of Re-search of the National Bureau of Standards 45, 4 (Oct. 1950), pp. 255-282.

[4] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, Emmanuel Thome, Joppe Bos, Pierrick Gaudry, Alexander Kruppa, Peter Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev and Paul Zimmermann, *Factorization of a 768-bit RSA modulus*, `http://eprint.iacr.org/2010/006`

[5] Flesh I., *A New Parallel Approach to the Block Lanczos Algorithm for Finding Nullspaces over GF(2)*, Master's thesis, 2002.

[6] Montgomery P., *Distributed Linear Algebra*, slides from 4th Workshop on Elliptic Curve Cryptography, 2000.

[7] J. Papadopoulos, *msieve*, `http://sourceforge.net/projects/msieve/`

[8] Hendrickson B., Leland R. and Plimpton S., *An Efficient Parallel Algorithm for Matrix-Vector Multiplication*, Int. J. High Speed Comput. 7(1), pp. 73-88, 1995.

[9] W. Eberly and E. Kaltofen, *On randomized Lanczos algorithms.* Proc. of the 1997 International Symposium on Symbolic and Algebraic Computation, ACM Press, 1997, pp. 176-183.

[10] Montgomery P.L. *A block Lanczos algorithm for finding dependencies over GF(2).*, in Proc. EUROCRYPT'95, vol. 921 of Springer Lecture Notes Comput. Sci., Springer-Verlag, pp. 106-120.

[11] *MPI Standard* `http://www.mcs.anl.gov/research/projects/mpi/`

[12] Chan E., Heimlich M., Purkayastha A., van de Geijn R.,*Collective Communication: Theory, Practice, and Experience*, Concurrency and Computation: Practice and Expirience, vol. 19, 13, pp 1749-1783, 2007.

[13] Knottenbelt W., Harrison P., *Efficient Parallel Sparse Matrix-Vector Multiplication Using Graph and Hypergraph Partitioning*, `http://www.doc.ic.ac.uk/~wjk/hypergraph-2up.pdf`, 2009.

E-mail address, I.A. Popovyan: `poilyard@gmail.com`