ACM Communications in Computer Algebra, Vol 47, No. 3, Issue 185, September 2013

Block Wiedemann algorithm on multicore architectures

Bastien Vialla
LIRMM, CNRS-UM2 France
bastien.vialla@lirmm.fr

Solving a linear system with large sparse matrices is a computational kernel used in a wide range of applications, *e.g.* cryptography, Gröbner basis ... Classical methods such as Gaussian elimination are not well suited because they tend to fill the matrix. In [7] Wiedemann proposed a blackbox algorithm which takes advantage of the sparsity to reduce the complexity. The main operations of this approach are sparse matrix vector products and the computation of the minimal generator of a scalar sequence. Despite a better complexity than classical methods, this algorithm is not efficient in the context of parallel computation as it needs a good repartition of the non-zero elements in the matrix. The block version of Wiedemann's algorithm proposed in [2] avoids this problem by using blocks instead of vectors. Therefore it offers parallelism outside the scope of the matrix.

Let $\mathbb{K}$ be a field, $A \in \mathcal{M}_{n\times n}(\mathbb{K})$, $U, V \in \mathcal{M}_{n\times k}(\mathbb{K})$ random matrices with $k \leq n$. We denote by γ the number of non-zero elements in A and we assume that $\gamma = O(n \log n)$. Block Wiedemann algorithm follows three steps:

1. Compute the first $O(\frac{2n}{k})$ elements of $S = [U^T A^i V]_{i\in\mathbb{N}}$ using $O(n\gamma + n^2 k^{\omega-2})$ operations in $\mathbb{K}$.
2. Find the minimal matrix polynomial generator of the sequence S using $\tilde{O}(k^{\omega-1}n)$ operations in $\mathbb{K}$.
3. Compute the solution using the polynomial found in step 2 using $O(n\gamma + n^2)$ operations in $\mathbb{K}$.

In practice the cost of the first step is dominant, therefore its parallelization is crucial. The capacity to parallelize the first step heavily relies on the dimension k of the blocks.

A classical approach is to take a block size equal to the number of cores. The parallel complexity of the first step becomes $O(\frac{n\gamma}{k} + n^2)$ operations in $\mathbb{K}$. We notice that the $O(n^2)$ part does not benefit from parallelism. In order to take advantage of parallelism everywhere in step 1, we must proceed otherwise.

Our approach We naturally extend the use of sparse blocks proposed by Eberly et al. in [3] to our context of block Wiedemann algorithm. Hence, instead of using random dense block for U, we use blocks of the form

$$U = \begin{bmatrix}\delta_1 I_k & \cdots & \delta_s I_k & \delta_{s+1} I'\end{bmatrix}^T \in \mathcal{M}_{n\times k}(\mathbb{K})$$

where $s = \lfloor n/k \rfloor$, $\delta_1, \cdots, \delta_{s+1} \in \mathbb{K}$ chosen at random, I_k the identity matrix of size k, and $I' = Diag(1, \cdots, 1) \in \mathcal{M}_{k\times r}(\mathbb{K})$ the matrix with only ones on the diagonal with $r = n \bmod k$. Using these new block projections, the sequential complexity of step 1 drops down to $O(n\gamma + n^2)$ operations in $\mathbb{K}$, eliminating the influence of the block size. In this work we study how these new block projections perform in practice and we show that they improve the performance of the first step of block Wiedemann algorithm.

Implementation and Benchmarks For the benchmarks, we have in mind matrices arising from NFS algorithm [6], which are very sparse. As γ is cheaper, the part of step 1 of complexity $O(n^2)$ has more importance. Therefore the block size has more influence on the sequence computation as γ is cheaper. In this case, we use a sparse matrix of size $10^5 \times 10^5$ over $\mathbb{F}_{65537}$ with ~ 15 non-zero elements per row uniformly dispatched.

The parallel complexity of step 1 using sparse blocks with k cores becomes $O(\frac{n\gamma+n^2}{k})$, hence offering perfect parallelism. So we want to see the influence of the block size on the computation of step 1.

First, we determine the most efficient block size depending on the number of cores. Let c be the number of cores, we benchmark the computation of the sequence starting with blocks of size c to $128c$ on 12 cores. As expected by the complexity analysis, a block size of c offers better performance for dense blocks. For sparse projections the theoretical study shows no influence of the blocks size. In practice we observe that a block size of $\simeq 32c$ is better,

which could be caused by memory management issues. However, this sparse block size is related to the number of non-zero elements of the matrix, so these values stand just for our test matrix. Despite their good complexity, sparse blocks have two flaws impacting the performance. The choice of matrix representation is important: first we choose to store blocks in column major representation to avoid concurrent writing, as suggested in [1]. Secondly, sparse blocks induce cache defaults as their size increase with the number of cores. To circumvent this problem, we permute block elements to obtain a cache friendly sparse blocks following ideas from [5]. For our tests we use an NUMA with four intel XEON E4620 with 8 cores at 2.2Ghz and 384GB of RAM. To obtain good performance on an NUMA, we design an hybrid MPI/tbb implementation that create one MPI process by node which use tbb to compute a part of the sequence. Each nodes own a copy of the sparse matrix and the block is split by column over the nodes, the results are gather at the end of the computation. All the libraries used are in the latest version from their svn directory. In table 1 we compare dense block which used LinBox's dense blocks implementation and our implementation using sparse blocks. For computing of dense block, LinBox relies on a BLAS library, in this case we use OpenBLAS wich is well optimized for intel XEON. The timings are in seconds and in parenthesis we indicate the block size used.

	Dense blocks (LinBox)		Sparse blocks	
	time in s	speed-up	time in s	speed-up
1 core	2205(1)	1	2165(32)	1
8 cores	540(8)	4	308(256)	7
16 cores	623(16)	3,5	154(512)	14
24 cores	798(24)	2,7	102(768)	21,2
32 cores	960(32)	2,2	77(1024)	28,1

Table 1: Times of sequence computation.

The time for dense blocks on one core is just for benchmark purposes. As predicted, sparse blocks perform better than dense blocks. However, the reasons that LinBox implementation does not perform well are that LinBox use an external library to compute dense block which as to create and destroy is own pool of threads for each computed element. Secondly, the LinBox implementation is not designed for a NUMA architecture as all the data is store in the first node memory.

This is a first step in an efficient implementation of block Wiedemann algorithm on multicore architectures. The next step will be an efficient implementation of $\sigma-$basis [4].

References

[1] Brice Boyer, Jean-Guillaume Dumas, and Pascal Giorgi. Exact Sparse Matrix-Vector Multiplication on GPUs and Multicore Architectures. *Proc. PASCO'10: Parallel Symbolic Computation*, 2010.

[2] Don Coppersmith. Solving Homogeneous Linear Equation Over GF(2) via Block Wiedemann Algorithm. *Mathematics of Computation*, 62(205):333–350, 1994.

[3] Wayne Eberly, Mark Giesbrecht, Pascal Giorgi, Arne Storjohann, and Gilles Villard. Faster inversion and other black box matrix computations using efficient block projections. *Proceedings of the 2007 international symposium on Symbolic and algebraic computation - ISSAC '07*, 3(1):143, 2007.

[4] Pascal Giorgi, Claude-Pierre Jeannerod, and Gilles Villard. On the complexity of polynomial matrix computations. *Proceedings of the 2003 international symposium on Symbolic and algebraic computation - ISSAC '03*, pages 135–142, 2003.

[5] Sardar A. Haque, Shahadat Hossain, and M. Moreno Maza. Cache friendly sparse matrix-vector multiplication. *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation (PASCO'10)*, pages 175–176, 2010.

[6] A. K. Lenstra and H. W. Lenstra. *The development of the number field sieve.* Springer-Verlag, 1993.

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