Parallelizing block Krylov iteration on multicore/manycore

Jérôme Javelle Simon Rousseau

ABSTRACT

Different ways are exposed to parallelize the computation of the characteristic polynomial of an $n \times n$ matrix over a field. Based on block Krylov iteration algorithm [1], they separate the Krylov subspace computation into parallel tasks in order to make good use of multicore architectures.

1. INTRODUCTION

Computing the characteristic polynomial of an $n \times n$ matrix over a field is a classical problem. Research has been done to make the algorithm as fast as possible, trying to tend to an asymptotic cost of $O(n^\theta)$ where θ is an admissible exponent for the complexity of matrix multiplication. C. Pernet and A. Storojohann [1] succeeded in making a Las Vegas randomized algorithm in exactly $O(n^\theta)$ field operations.

Still, these are sequential algorithms which don't make good use of the parallel computing architectures we find in modern computers. We'd like to figure out some ways of isolating tasks that could be executed in parallel, to make the computation faster in such architectures.

In this paper, we put a further look at [1] in order to identify the steps that can be efficiently parallelized. Then we study different schemes of parallelization, making previsions and testing them with several machines to compare their performances.

2. OVERVIEW OF THE ALGORITHM

The aim of the global algorithm is to compute the characteristic polynomial of a matrix `A` of size `N` using iterated blocs Krylov.

We pick a random matrix of size $N.noc$ where $noc = \frac{n}{c}$ (`c` is the number of iterations). Then we compute the re-ordered Krylov subspace

$$\left[u_1, Au_1, ..., A^{c-1}u_1, ..., ..., u_{noc}, Au_{noc}, ..., A^{c-1}u_{noc}\right]$$

After some substitutions and permutations, we get a shifted Hessenberg form for the Krylov matrix :

$$\begin{bmatrix} C_1 & B_* & \cdots & B_* & B_* & B_* & \cdots & B_* \\ B_* & C_2 & \cdots & B_* & B_* & B_* & \cdots & B_* \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ B_* & B_* & \cdots & C_j & B_* & B_* & \cdots & B_* \\ & & & & C_{j+1} & B_* & \cdots & B_* \\ & & & & B_* & C_{j+2} & \cdots & B_* \\ & & & & & & \ddots & \\ & & & & B_* & B_* & \ddots & B_* \\ & & & & B_* & B_* & \cdots & C_m \end{bmatrix}$$

The computation of the characteristic polynomial is immediate, since the Krylov matrix is block upper triangular, with blocks of the form :

$$\begin{bmatrix} 0 & \cdots & 0 & * \\ 1 & \ddots & \vdots & \vdots \\ & \ddots & 0 & \vdots \\ & & 1 & * \end{bmatrix}$$

This characteristic polynomial is the product of the polynoms of the companion blocks, whose coefficients are the opposite of the numbers in the last column.

The part of this algorithm that we implement is the construction of the Krylov subspace for a matrix `A` and a random matrix `U`. The degrees of freedom for the algorithm are the number of threads used, the number `c` of iterations in the Krylov subspace.

We will try to find the optimal parmeters for each algorithm used through experimentations on a 16 core system.

3. PARALLELISATION SCHEMES

3.1 Subdivising the iterated submatrix A^kU

We have to compute the Krylov matrix

$$\left[u_1, Au_1, \ldots, A^{c-1}u_1, \ldots, \ldots, u_{noc}, Au_{noc}, \ldots, A^{c-1}u_{noc}\right]$$

using a matrix U of size $noc \times N$ and the matrix `A`. The first parallelization scheme consists in dividing the matrix `U` into several submatrix U_i. Each thread calls a routine that computes the matrix $\left[U_i, AU_i, \ldots, A^{c-1}U_i\right]$.

An advantage of such a parallelization is that the synchronization can occur only at the end of the whole computation of the Krylov matrix.

The asynchronous characteristic of this subdivision provides a good efficiency as regards the computing time.

The point which can be discussed is the following : dividing the matrix `U` which is rectangular involves the parallel computations of matrix multiplication of size $N \times N \times size\_submatrix$, where `size_submatrix` is the size of the matrix U_i. The size `size_submatrix` is a lot smaller than `N`, therefore the matrix multiplication is unbalanced, which involves a bigger use of memory (N^3 operations for N^3 memory units for `N` multiplications matrix/vector, against N^3 operations for N^2 memory units for a multiplication of square matrix).

We tried to use another parallelization scheme, which consists in the subdivision of the matrix `A`.

3.2 Subdivising the matrix A

Computing the Krylov matrix described in the previous subsection can be done by subdivising the matrix `A` like this:

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_{nb\_threads} \end{bmatrix}$$

Indeed, the dimensions of the matrix multiplications are $N * \frac{N}{nb\_threads} * noc$, which is less unbalanced, therefore more efficient as regards memory.

In this algorithm, the matrix A^kU is also computed from the matrix $A^{k-1}U$, then we must wait for the computation of the previous step to be able to go one step further; consequently, it requires a synchronization at each step, which represents some cost. The lower the number of iterations is, the less synchronizations there are, and the more efficient the algorithm is. Thus, lower values of the parameter `c` will give an optimal computing time in this case.

3.3 Hybrid Method

The combination of both methods can be imagined in order to have take advantage of all positive aspects.
Then, it will be more difficult to get results from experimentation since there are new parameters : the repartition of the threads between the first and the second version of the algorithm.

4. RESULTS

4.1 Subdivising the iterated submatrix A^kU

The tests are run on `ensibm.imag.fr`, a 16 core system. This explains the lack of efficiency obtained for a number of threads close to 16.

On these graphics, we can see a good evolution of the computing time with a growing number of threads, and our programm is close to the ideal line, except for a number of threads close to 16. This shows that for a large enough matrix `A`(size 2000 here), the establishement of the multithreading doesn't take a significant time. [Fig. 1] [Fig. 2]

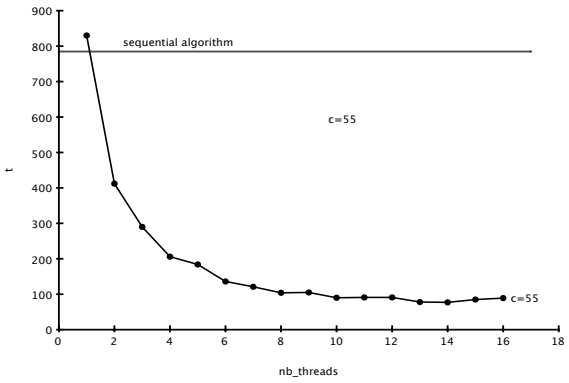

Figure 1: Variation of duration with the number of threads

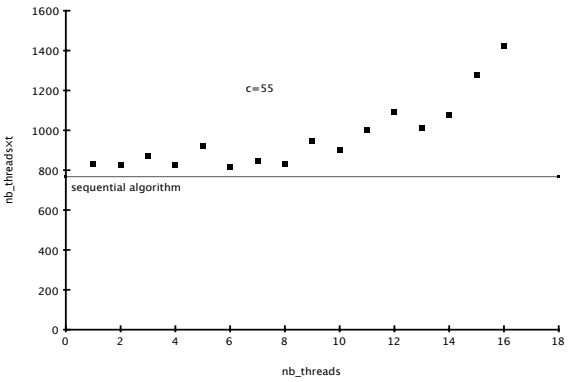

Figure 2: Variation of duration with the number of threads

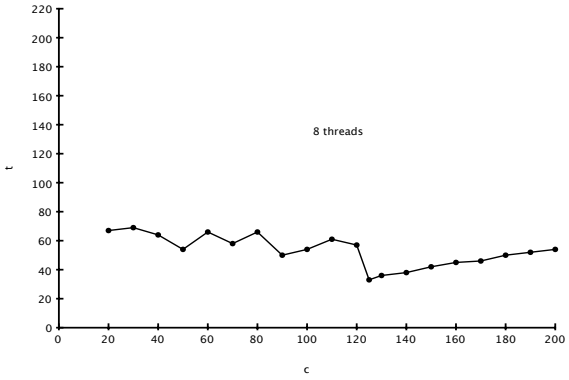

Figure 3: 8 threads

This kind of experiment is made to find an optimal parameter c for a fixed number of thread. Indeed, if we want to use 8 threads (for example), we must know the value of `c` which gives us the shortest computating time.

For the case of 8 threads [Fig. 3], the optimal parameter `c` is around 125. Before that, the computating time is not regular because of the subdivisions of the matrix A^kU.

After the optimal value, the computing time is linearly in-

creasing, like in the sequential case.

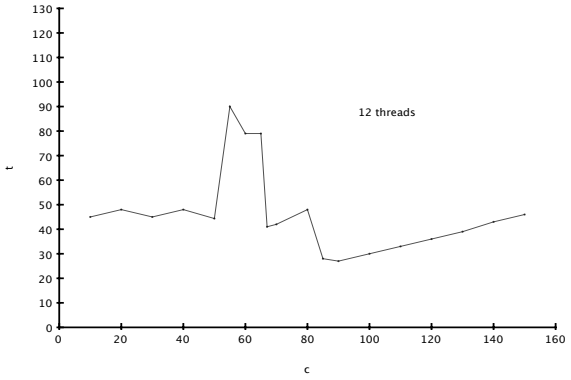

Figure 4: 12 threads

Fig. 4 shows the optimal value of c for 12 threads seems to be around 90. We observe the same phenomenon than in the previous case before and after this value.

4.2 Subdivising the matrix *A*

Experiments on matrix of different sizes shows that, for a fixed number of threads, the optimal value of `c` is rather high. This shows that having a good balance of dimensions in the matrix multiplication brings more efficiency than worrying about synchronization of the threads.

Even if this programm is not totally asynchronous, the subdivision of the matrix `A` is regular enough to reduce the consequences of the synchronization.

5. ONGOING WORKS

5.1 Parallelizing with CILK

CILK is a programming language for multithreaded parallel computing, based on C. It's especially effective for exploiting dynamic, highly asynchronous parallelism, so it would be particularly interesting to implement our parallel algorithm using CILK.

There is a C++ version of CILK called CILK++, under commercial license. It's not possible to integrate a CILK / CILK++ source code into a C++ program, so we needed to start from scratch. However, as we just had time to reuse the source code that came from the original sequential program, we couldn't make a whole new program in CILK or CILK++.

5.2 Parallelizing with KAAPI

The KAAPI library offers a parallel programming interface for distributed computing. The interface is called Athapascan, it's a high-level language, C++ compatible, based on the separation of the algorithm into *tasks*. Athapascan works with a global memory space, the idea is to define tasks, shared variables and the dependencies between tasks.

KAAPI is particularly well designed for distributed computing, so it would have been interesting to implement a KAAPI version of the algorithm and to test it with a network of several computers. We tested some basic KAAPI programs and the library appeared to be very effective for parallel computing. Unfortunately we didn't have enough time to adapt our algorithm to this environment.

6. CONCLUSION

This programm and the experiments made on the different ways to divide the problem showed that the computation of a characteristic polynomial is likely to be adapted to multicore/manycore architectures.

Continuing this projet with other methods to divide the computations might offer a new way for the factorization of great numbers.

7. REFERENCES

[1] C. Pernet and A. Storjohann. *Faster Algorithms for the Characteristic Polynomial*. In Proc. *ISSAC '07*, pages 307-314. ACM Press, New York, 2007.

[2] D. Augot and P. Camion. *Forme de Frobenius et vecteurs cycliques*. Comptes-rendus de l'Académie des Sciences, Paris, t. 318, Série I, pages 183-188, 1994.

[3] S. Guelton, *Athapascan Overview*. KAAPI Library website http://kaapi.gforge.inria.fr/doc/atha.html.

[4] M. Frigo, *The CILK Project*. CILK Library website http://supertech.csail.mit.edu/cilk/.