Two parallel implementations of Ehrlich-Aberth algorithm for root-finding of polynomials on multiple GPUs with OpenMP and MPI

Kahina Ghidouche, Abderrahmane Sider, Lilia Ziane Khodja, Raphael Couturier

Two parallel implementations of Ehrlich-Aberth algorithm for root-finding of polynomials on multiple GPUs with OpenMP and MPI

Kahina Ghidouche, Abderrahmane Sider
Laboratoire LIMED
Faculté des sciences exactes
Université de Bejaia, 06000, Algeria
Email: {kahina.ghidouche,ar.sider}@univ-bejaia.dz

Lilia Ziane Khodja, Raphaël Couturier
FEMTO-ST Institute
University Bourgogne Franche-Comte, France
Email: zianekhodja.lilia@gmail.com
raphael.couturier@univ-fcomte.fr

***Abstract*—Finding the roots of polynomials is a very important part of solving real-life problems but the higher the degree of the polynomials is, the less easy it becomes. In this paper, we present two different parallel algorithms of the Ehrlich-Aberth method to find roots of sparse and fully defined polynomials of high degrees. Both algorithms are based on CUDA technology to be implemented on multi-GPU computing platforms but each use different parallel paradigms: OpenMP or MPI. The experiments show a quasi-linear speedup by using up-to 4 GPU devices compared to 1 GPU to find the roots of polynomials of degree up-to 1.4 million. Moreover, other experiments show it is possible to find the roots of polynomials of degree up-to 5 million.**

***Index Terms*—root finding method, Ehrlich-Aberth method, GPU, MPI, OpenMP**

I. Introduction

Finding the roots of polynomials of very high degrees arises in many complex problems of various domains such as algebra, biology or physics. A polynomial $p(x)$ in $\mathbb{C}$ in one variable x is an algebraic expression in x of the form:

$$p(x) = \sum_{i=0}^{n} \alpha_i x^i, \alpha_n \neq 0, \tag{1}$$

where $\{\alpha_i\}_{0 \leq i \leq n}$ are complex coefficients and n is a high integer number. If $\alpha_n \neq 0$ then n is called the degree of the polynomial. The root-finding problem consists in finding the n different values of the unknown variable x for which $p(x) = 0$. Such values are called roots of $p(x)$. Let $\{z_i\}_{1 \leq i \leq n}$ be the roots of polynomial $p(x)$, then $p(x)$ can be written as :

$$p(x) = \alpha_n \prod_{i=1}^{n} (x - z_i), \alpha_n \neq 0. \tag{2}$$

Most of the numerical methods that deal with the polynomial root-finding problems are simultaneous methods, *i.e.* the iterative methods to find simultaneous approximations of the n polynomial roots. These methods start from the initial approximation of all n polynomial roots and give a sequence of approximations that converge to the roots of the polynomial. Two examples of well-known simultaneous methods for root-finding problem of polynomials are the Durand-Kerner method [1], [2] and the Ehrlich-Aberth method [3], [4].

The convergence time of simultaneous methods drastically increases with the increasing of the polynomial's degree. The great challenge with simultaneous methods is to parallelize them and to improve their convergence. Many authors have proposed parallel simultaneous methods [5], [6], [7], [8], [9], [10], using several paradigms of parallelization (synchronous or asynchronous computations, mechanism of shared or distributed memory, etc). However, so far until now, only polynomials not exceeding degrees of less than 100,000 have been solved.

The recent advent of the Compute Unified Device Architecture (CUDA) [11], a programming model and a parallel computing architecture developed by NVIDIA, has revived parallel programming interest in this problem. Indeed, the computing power of GPUs (Graphics Processing Units) has exceeded that of traditional CPUs processors, which makes it very appealing to the research community to investigate new parallel implementations for a whole set of scientific problems in the reasonable hope to solve bigger instances of well known computationally demanding issues such as the one beforehand. However, CUDA provides an efficient massive data computing model which is suited to GPU architectures. Ghidouche et al. [12] proposed an implementation of the Durand-Kerner method on a single GPU Tesla 2070. Their main results showed that a parallel CUDA implementation is about 10 times faster than the sequential implementation on a single CPU Intel(R) Xeon(R) CPU E5620@2.40GHz for sparse polynomials of degree 48,000.

In this paper we propose the parallelization of the Ehrlich-Aberth (EA) method which has a much better cubic convergence rate than the quadratic rate of the Durand-Kerner method that has already been investigated in [12]. In the other hand, EA is suitable to be implemented in parallel computers according to the data-parallel paradigm. In this model, computing elements carry computations on the data they are assigned and communicate with other computing elements in order to get fresh data or to synchronize. Classi-

cally, two parallel programming paradigms OpenMP and MPI are used to code such solutions. But in our case, computing elements are CUDA multi-GPU platforms. This architectural setting poses new programming challenges but offers also new opportunities to efficiently solve huge problems, otherwise considered intractable until recently. To the best of our knowledge, our CUDA-MPI and CUDA-OpenMP codes are the first implementations of EA method with multiple GPUs for finding roots of polynomials. Our major contributions include:

- The parallel implementation of EA algorithm on a multi-GPU platform with a shared memory using OpenMP API. It is based on threads created from the same system process, such that each thread is attached to one GPU. In this case the communications between GPUs are done by OpenMP threads through shared memory.
- The parallel implementation of EA algorithm on a multi-GPU platform with a distributed memory using MPI API, such that each GPU is attached and managed by a MPI process. The GPUs exchange their data by message-passing communications. This approach is more used on clusters to solve very complex problems that are too large for traditional supercomputers, which are very expensive to build and run.
- Our method is efficient to compute the roots of sparse and full polynomials of degree up to 5 million.

The paper is organized as follows. In Section II we present three different parallel programming models OpenMP, MPI and CUDA. In Section III we present the implementation of the Ehrlich-Aberth algorithm on a single GPU. In Section IV we present the parallel implementations of the Ehrlich-Aberth algorithm on multiple GPUs using the OpenMP and MPI approaches. In section V we present our experiments and discuss them. Finally, Section VI concludes this paper and gives some hints for future research directions in this topic.

II. Parallel programming models

Our objective consists in implementing a root-finding algorithm of polynomials on multiple GPUs. To this end, it is essential to know how to manage the CUDA contexts of different GPUs. A direct method to control the various GPUs is to use as many threads or processes as GPU devices. We investigate two parallel paradigms: OpenMP and MPI. In this case, the GPU indices are defined according to the identifiers of the OpenMP threads or the ranks of the MPI processes. In this section we present the parallel programming models: OpenMP, MPI and CUDA.

A. OpenMP

OpenMP (Open Multi-processing) is an application programming interface for parallel programming [13]. It is a portable approach based on the multithreading designed for shared memory computers, where a master thread forks a number of slave threads which execute blocks of code in parallel. An OpenMP program alternates sequential regions and parallel regions of code, where the sequential regions are executed by the master thread and the parallel ones may be executed by multiple threads. During the execution of an OpenMP program the threads communicate their data (read and modified) in the shared memory. One advantage of OpenMP is the global view of the memory address space of an application. This allows a relatively fast development of parallel applications with easier maintenance. However, it is often difficult to get high rates of performances in large scale-applications.

B. MPI

MPI (Message Passing Interface) is a portable message passing style of the parallel programming designed specifically for distributed memory architectures [14]. In most MPI implementations, a computation contains a fixed set of processes created at the initialization of the program in such a way that one process is created per processor. The processes synchronize their computations and communicate by sending/receiving messages to/from other processes. In this case, the data are explicitly exchanged by message passing while the data exchanges are implicit in a multithread programming model like OpenMP and Pthreads. However in the MPI programming model, the processes may either execute different programs referred to as multiple program multiple data (MPMD) or every process executes the same program (SPMD). The MPI approach is one of the most used HPC programming model to solve large scale and complex applications.

C. CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA [11] for GPUs. It provides a high level GPGPU-based programming model to program GPUs for general purpose computations. The GPU is viewed as an accelerator such that data-parallel operations of a CUDA program running on a CPU are off-loaded onto GPU and executed by this latter. The data-parallel operations executed by GPUs are called kernels. The same kernel is executed in parallel by a large number of threads organized in grids of thread blocks, such that each GPU multiprocessor executes one or more thread blocks in SIMD fashion (Single Instruction, Multiple Data) and in turn each core of the multiprocessor executes one or more threads within a block. Threads within a block can cooperate by sharing data through a fast shared memory and coordinate their execution through synchronization points. In contrast, within a grid of thread blocks, there is no synchronization at all between blocks. The GPU only works on data filled in the global memory and the final results of the kernel executions must be transferred out of the GPU. In the GPU, the global memory has lower bandwidth than the shared memory associated to each multiprocessor. Thus with CUDA programming, it is necessary to design carefully the arrangement of the thread blocks in order to ensure a low latency and a proper use of the shared memory. As for the global memory accesses, it should also be minimized.

III. The Ehrlich-Aberth algorithm on a GPU

A. The Ehrlich-Aberth method

The Ehrlich-Aberth method is a simultaneous method [4] using the following iteration

$$z_i^{k+1} = z_i^k - \frac{\frac{p(z_i^k)}{p'(z_i^k)}}{1 - \frac{p(z_i^k)}{p'(z_i^k)} \sum_{j=1, j\neq i}^{j=n} \frac{1}{(z_i^k - z_j^k)}}, i = 1, \ldots, n \quad (3)$$

This method contains 4 steps. The first step consists in the initializing the polynomial. The second step initializes the solution vector Z using the Guggenheimer method [15] to ensure that initial roots are all distinct from each other. In step 3, the iterative function based on the Newton's method [16] and Weiestrass operator [17] is applied. In our case, the Ehrlich-Aberth is applied as in (3). Iterations of the EA method will converge to the roots of the considered polynomial. In order to stop the iterative function, a stop condition is applied, this is the 4th step. This condition checks that all the root modules are lower than a fixed value ϵ.

$$\forall i \in [1, n], \ |\frac{z_i^k - z_i^{k-1}}{z_i^k}| < \epsilon \quad (4)$$

B. Improving Ehrlich-Aberth method

With high degree polynomials, the EA method suffers from floating point overflows due to the mantissa of floating points representations. This induces errors in the computation of $p(z)$ when z is large.

In order to solve this problem, we propose to modify the iterative function by using the logarithm and the exponential of a complex and we propose a new version of the EA method. This method allows us to exceed the computation of the polynomials of degree 100,000 and to reach a degree up to more than 1,000,000. The reformulation of the iteration (3) of the EA method with exponential and logarithm operators is defined as follows, for $i = 1, \ldots, n$:

$$z_i^{k+1} = z_i^k - \exp(\ln(p(z_i^k)) - \ln(p'(z_i^k)) - \ln(1 - Q(z_i^k))), \quad (5)$$

where:

$$Q(z_i^k) = \exp(\ln(p(z_i^k)) - \ln(p'(z_i^k)) + \ln(\sum_{i\neq j}^{n} \frac{1}{z_i^k - z_j^k})). \quad (6)$$

Using the logarithm and the exponential operators, we can replace any multiplications and divisions with additions and subtractions. Consequently, computations manipulate lower values in absolute values [18]. In practice, the exponential and logarithm mode is used when a root is outside the circle unit represented by the radius R evaluated in C language with:

$$R = exp(log(DBL\_MAX)/(2 * n)); \quad (7)$$

where `DBL_MAX` stands for the maximum representable `double` value and n is the degree of the polynomial.

C. The Ehrlich-Aberth parallel implementation on CUDA

The algorithm 1 shows sketch of the Ehrlich-Aberth method using CUDA. The first step consists in the initialization of the input data, for exemple the polynomial P, the derivative of P and the vector solution Z. Then, all data of the root finding problem are transfered from the CPU memory to the GPU global memory, because the GPUs only work on the data filled in their memories. Next, all the data-parallel arithmetic operations inside the main loop `(while(...))` are executed as kernels by the GPU. The first kernel named *Kernelsave* in line 5 of Algorithm 1 consists in saving the vector of polynomial roots found at the previous time-step in GPU memory, in order to check the convergence of the roots after each iteration (line 7, Algorithm 1). Then the new roots with the new iterations are computed using the EA method with a Gauss-Seidel iteration mode in order to use the latest updated roots (line 6). This improves the convergence compared to the Jacobi method. This kernel is, in practice, very long since it performs all the operations with complex numbers with the normal mode of the EA method as in Eq. 3 but also with the logarithm-exponential one as in Eq.(6, 5). The last kernel checks the convergence of the roots after each update of Z^k, according to formula Eq. 4 line (7). We used the functions of the CUBLAS Library (CUDA Basic Linear Algebra Subroutines) to implement this kernel. The algorithm terminates its computations when all the roots have converged.

Algorithm 1: Finding roots of polynomials with the Ehrlich-Aberth method on a GPU

Input: ϵ (tolerance threshold)
Output: Z (solution vector of roots)
1 Initialize the polynomial P and its derivative P';
2 Set the initial values of vector Z;
3 Copy P, P' and Z from CPU to GPU;
4 **while** $error > \epsilon$ **do**
5 $\quad Z^{prev}$ = KernelSave(Z);
6 $\quad Z$ = KernelUpdate(P, P', Z);
7 $\quad error$ = KernelComputeError(Z, Z^{prev});
8 Copy Z from GPU to CPU;

Listing 1 shows the a simplified version of second kernel code (some parameters in the kernels have been simplified in order to increase the readability). As can be seen this kernel calls multiple kernels, all the kernels for complex numbers and kernels for the evaluation of a polynomial are not detailed.

Listing 1. Kernels to update the roots

```
//Normal version of the Ehrlich-Aberth method
__device__
cuDoubleComplex FirstH_EA(int i, cuDoubleComplex *Z) {

  cuDoubleComplex  result;
  cuDoubleComplex  C,F,Fp;
  int              j;
  cuDoubleComplex  sum;
  cuDoubleComplex  un;
```

```
  //evaluate the polynomial
  F = Fonction(Z[i]);
  //evaluate the derivative of the poly.
  Fp=FonctionD(Z[i]);

  double mod=Cmodule(F);
  sum.x=0;sum.y=0;
  un.x=1;un.y=0;
  C=Cdiv(F,Fp);        //P(z)/P'(z)

  //for all roots, compute the sum
  //for the Ehrlich-Aberth iteration
  for ( j=0 ; j<P.PolyDegre ; j++ )
  {
    if ( i != j )
    {
      sum=Cadd(sum,Cdiv(un,Csub(Z[i],Z[j])));
    }
  }
  sum=Cdiv(C,Csub(un,Cmul(C,sum)));    //C/(1-Csum)
  result=Csub(Z[i], sum);
  return (result);
}


//Log Exp version of the Ehrlich-Aberth method
__device__
cuDoubleComplex NewH_EA(int i, cuDoubleComplex *Z) {

  cuDoubleComplex result;
  cuDoubleComplex F,Fp;
  cuDoubleComplex one,denominator,sum;
  int             j;
  one.x=1;one.y=0;
  sum.x=0;
  sum.y=0;

  //evaluate the polynomial with
  //the LogExp version
  Fp = LogFonctionD(Z[i]);
  //evaluate the derivative of the polynomial
  //with the LogExp version
  F = LogFonction(Z[i]);

  cuDoubleComplex FdivFp=Csub(F,Fp);

  //for all roots, compute the sum
  //for the Ehrlich-Aberth iteration
  for ( j=0 ; j<P.degrePolynome ; j++ )
  {
    if ( i != j )
    {
      sum=Cadd(sum,Cdiv(un,Csub(Z[i],Z[j])));
    }
  }

  //then terminate the computation
  //of the Ehrlich-Aberth method
  denominator=Cln(Csub(un,Cexp(Cadd(FdivFp,Cln(sum)))));
  result=Csub(FdivFp,denominator);
  result=Csub(Z[i],Cexp(res));

  return   result;

}


//kernels to update a root i
__global__
void Dev_EA(int i, cuDoubleComplex *Z, int* finished,
            int size) {
  int i= blockIdx.x*blockDim.x+ threadIdx.x;
  if(i<size) {
    //if the root needs to be updated
    if(!finished[i]) {
      //according to the module of the root
      if (Cmodule(Z[i])<=maxRadius)
        //selects the normal version
        Z[i] = FirstH_EA(i,Z);
      else
        //of the Log Exp version
        Z[i] = NewH_EA(i,Z);
      return c;
    }
  }
}
```

The development of this code is a rather long task due to the development of all the kernels that compute the parts ported on the GPU. This comes in particular from the fact that it is very difficult to debug CUDA running threads like threads on a CPU host. In the following section the GPU parallel implementation of the Ehrlich-Aberth method with OpenMP and MPI is presented.

IV. The Ehrlich-Aberth algorithm on multiple GPUs

In order to manage the CUDA contexts of different GPUs, two parallel paradigms are investigated: OpenMP and MPI. In this section we present the *OpenMP-CUDA* and the *MPI-CUDA* approaches used to implement the Ehrlich-Aberth algorithm on multiple GPUs.

A. An OpenMP-CUDA approach

Our OpenMP-CUDA implementation of EA algorithm is based on the hybrid OpenMP and CUDA programming model. This algorithm is presented in Algorithm 2. All the data are shared with OpenMP among all the OpenMP threads. The shared data are the solution vector Z, the polynomial to solve P, its derivative P', and the error vector $error$. The number of OpenMP threads is equal to the number of GPUs, each OpenMP thread binds to one GPU, and it controls a part of the shared memory. More precisely each OpenMP thread will be responsible for updating its own part of the vector Z. This part is called Z_{loc} in the following. Then all GPUs will have a grid of computation organized according to the device performance and the size of data on which it runs the computation kernels.

To compute one iteration of the EA method each GPU performs the followings steps. First, roots are shared with OpenMP and the computation of the local size for each GPU is performed (line 4). Each thread starts by copying all the previous roots inside its GPU (line 5). At each iteration, the following operations are performed. First the vector Z is transferred from the CPU to the GPU (line 7). Each GPU copies the previous roots (line 8) and it computes an iteration of the EA method on its own roots (line 9). For that all the other roots are used. The local error is computed on the new roots (line 10) and the maximum of the local errors is computed on all OpenMP threads (line 11). At the end of an iteration, the updated roots are copied from the GPU to the CPU (line 12) and each CPU directly updates its own roots in the shared memory arrays containing all the roots.

Algorithm 2: Finding roots of polynomials with the Ehrlich-Aberth method on multiple GPUs using OpenMP

Input: ϵ (tolerance threshold)
Output: Z (solution vector of roots)

1 Initialize the polynomial P and its derivative P';
2 Set the initial values of vector Z;
3 Start of a parallel part with OpenMP (Z, $error$, P, P' are shared variables);
4 Determine the local part of the OpenMP thread;
5 Copy P, P' from CPU to GPU;
6 **while** $error > \epsilon$ **do**
7 Copy Z from CPU to GPU;
8 Z_{loc}^{prev} = KernelSave(Z_{loc});
9 Z_{loc} = KernelUpdate(P, P', Z);
10 $error_{loc}$ = KernelComputeError(Z_{loc}, Z_{loc}^{prev});
11 $error = max(error_{loc})$;
12 Copy Z_{loc} from GPU to Z in CPU;

Algorithm 3: Finding roots of polynomials with the Ehrlich-Aberth method on multiple GPUs using MPI

Input: ϵ (tolerance threshold)
Output: Z (solution vector of roots)

1 Initialize the polynomial P and its derivative P';
2 Set the initial values of vector Z;
3 Determine the local part of the MPI process;
4 Computation of the parameters for the $MPI\_AlltoAll$;
5 Copy P, P' from CPU to GPU;
6 **while** $error > \epsilon$ **do**
7 Copy Z from CPU to GPU;
8 Z_{loc}^{prev} = KernelSave(Z_{loc});
9 Z_{loc} = KernelUpdate(P, P', Z);
10 $error_{loc}$ = KernelComputeError(Z_{loc}, Z_{loc}^{prev});
11 $error = MPI\_Reduce(error_{loc})$;
12 Copy Z_{loc} from GPU to CPU;
13 $Z = MPI\_AlltoAll(Z_{loc})$;

B. A MPI-CUDA approach

Our parallel implementation of EA to find the roots of polynomials using a CUDA-MPI approach follows a similar approach to the one used in CUDA-OpenMP. Each processor is responsible for computing its own part of roots using all the roots computed by other processors at the previous iteration. The difference between both approaches lies in the way processors communicate and exchange data. With MPI, processors need to send and receive data explicitly. So in Algorithm 3, after the initialization phase all the processors have the same Z vector. Then they need to compute the parameters used by the $MPI\_AlltoAll$ routines (line 4). In practice, each processor needs to compute its offset and its local size. Processors need to allocate memory on their GPU and need to copy their data on the GPU (line 5). At the beginning of each iteration, a processor starts by transferring the whole vector Z from the CPU to the GPU (line 7). Only the local part of Z^{prev} is saved (line 8). After that, a processor is able to compute an updated version of its own roots (line 9) with the EA method. The local error is computed (line 10) and the global error is also computed using MPI_Reduce (line 11). Then the local roots are transferred from the GPU memory to the CPU memory (line 12) before being exchanged between all processors (line 13) in order to give to all processors the last version of the roots (with the $MPI\_AlltoAll$ routine). If the convergence is not satisfied, a new iteration is executed.

V. Experiments

We study two categories of polynomials: sparse polynomials and full polynomials.

A sparse polynomial is a polynomial for which only some coefficients are not null. In this paper, we consider sparse polynomials for which the roots are distributed on 2 distinct circles:

$$\forall \alpha_1 \alpha_2 \in \mathbb{C}, \forall n_1, n_2 \in \mathbb{N}^*; p(z) = (z^{n_1} - \alpha_1)(z^{n_2} - \alpha_2) \quad (8)$$

A full polynomial is, in contrast, a polynomial for which all the coefficients are not null. A full polynomial is defined by:

$$\forall \alpha_i \in \mathbb{C}, i \in \mathbb{N}; p(x) = \sum_{i=0}^{n} \alpha_i . x^i \quad (9)$$

In the following experiments, all the reported results have been obtained on a machine equipped with a CPU Intel(R) Xeon(R) CPU X5620@2.40GHz node with 64GB of ram and 4 Tesla Kepler K40 GPUs (2880 cores) with CUDA 7.5, OpenMP and the OpenMPI 1.7.5.

In order to evaluate both the GPU and Multi-GPU approaches, we performed a set of experiments on a single GPU and multiple GPUs using OpenMP or MPI with the EA algorithm, for both sparse and full polynomials of different degrees. All experimental results obtained are performed with double precision floating-point data and the convergence threshold of the EA method is set to 10^{-7}. The initialization values of the vector solution of the methods are given by the Guggenheimer method [15].

A. Evaluation of the multi-GPUs approaches

In this part, we evaluate the performances of the CUDA-OpenMP and CUDA-MPI approaches of the EA algorithm on different GPU platforms composed each of 1, 2, 3 or 4 GPUs. In this experiments we report the experimental results of the EA algorithms to find the roots of different sparse and full polynomials of high degrees ranging from 100,000 to 1,400,000. Figures 1 and 2 show the execution times to solve, respectively, sparse and full polynomials with the CUDA-OpenMP algorithm, and Figures 3 and 4 show those to solve, respectively, sparse and full polynomials with the CUDA-MPI algorithm.

All these figures show that the CUDA-OpenMP and the CUDA-MPI approaches of the EA algorithm, compared to the single GPU version, are efficient and scale well with multiple

GPUs. Both approaches allow us to solve sparse and full polynomials of very high degrees. Using 4 GPUs allows us to achieve a quasi-linear speedup.

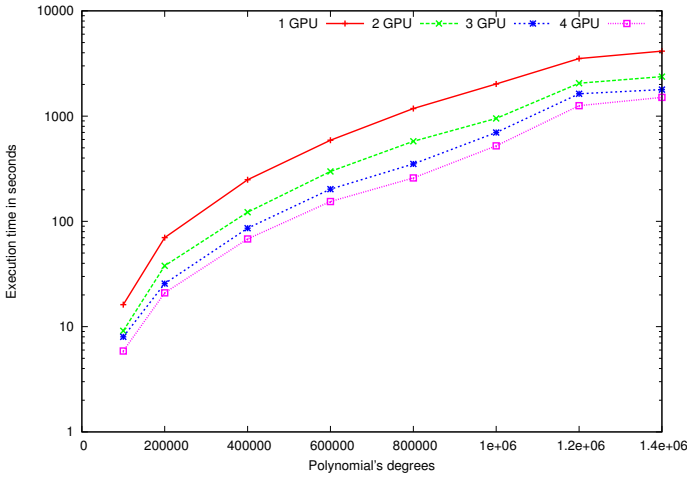

Fig. 1. Execution times in seconds of the Ehrlich-Aberth method to solve sparse polynomials on multiple GPUs with CUDA-OpenMP.

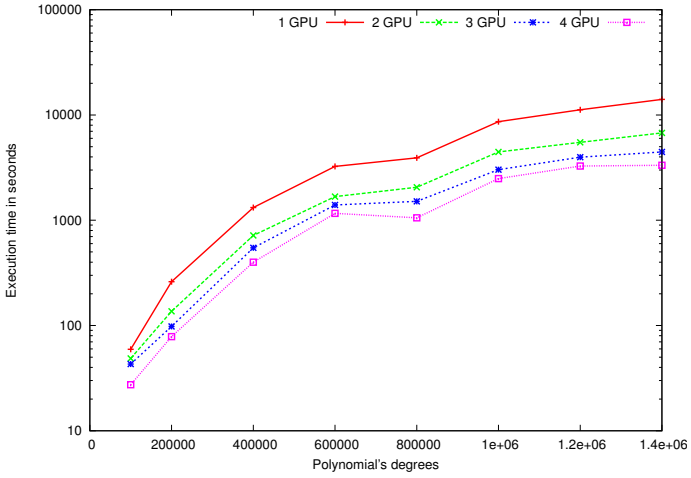

Fig. 2. Execution times in seconds of the Ehrlich-Aberth method to solve full polynomials on multiple GPUs with CUDA-OpenMP.

B. Comparison between the CUDA-OpenMP and the CUDA-MPI approaches

In the previous section we saw that both approaches are very efficient to reduce the execution times to solve sparse and full polynomials. In this section we try to compare these two approaches. In this experiment three sparse polynomials and three full polynomials of degrees 200,000, 800,000 and 1,400,000 are investigated. Figures 5 and 6 show the comparison between CUDA-OpenMP and CUDA-MPI algorithms of the EA method to solve sparse and full polynomials, respectively.

In Figure 5 there is one curve for CUDA-OpenMP and another one for CUDA-MPI for each polynomial investigated. We can see that the results are quite similar between OpenMP

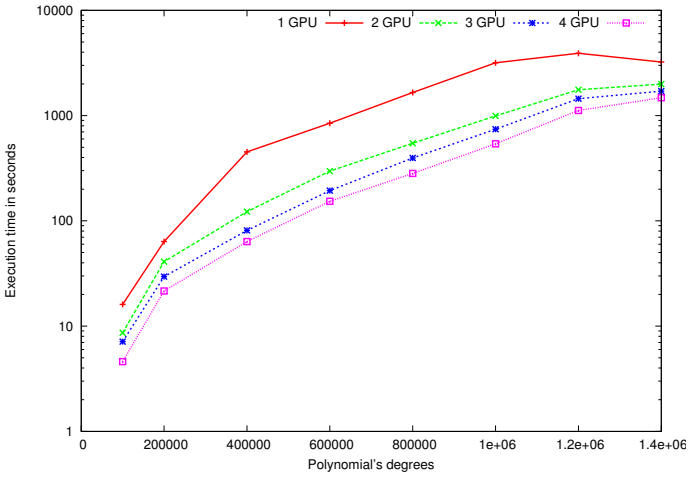

Fig. 3. Execution times in seconds of the Ehrlich-Aberth method to solve sparse polynomials on multiple GPUs with CUDA-MPI.

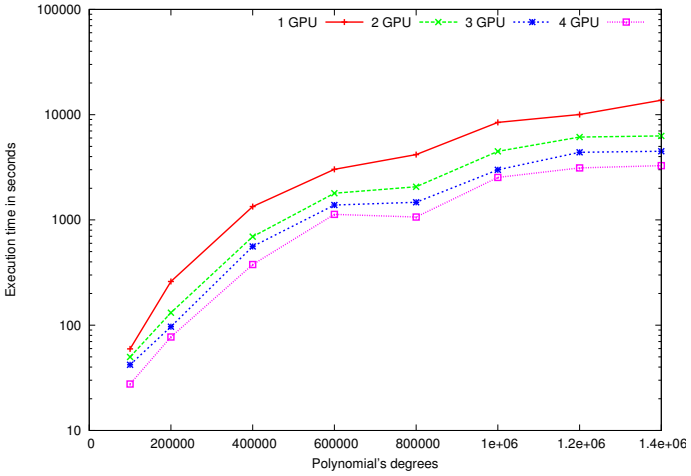

Fig. 4. Execution times in seconds of the Ehrlich-Aberth method for full polynomials on multiple GPUs with CUDA-MPI.

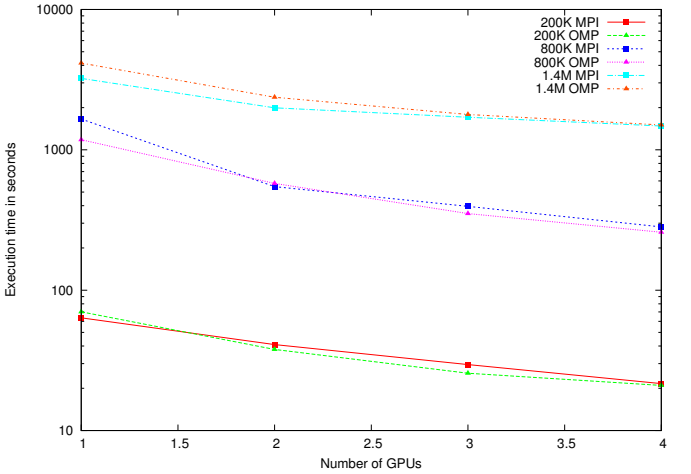

Fig. 5. Execution times to solve sparse polynomials of three distinct degrees on multiple GPUs using OpenMP and MPI with the Ehrlich-Aberth method

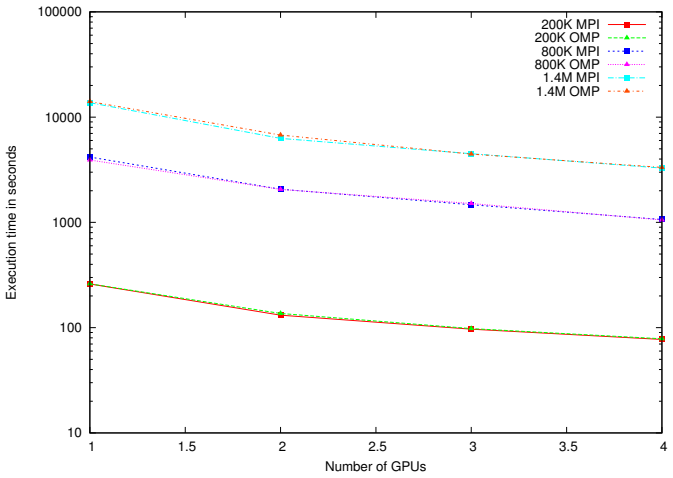

Fig. 6. Execution times to solve full polynomials of three distinct degrees on multiple GPUs using OpenMP and MPI with the Ehrlich-Aberth method

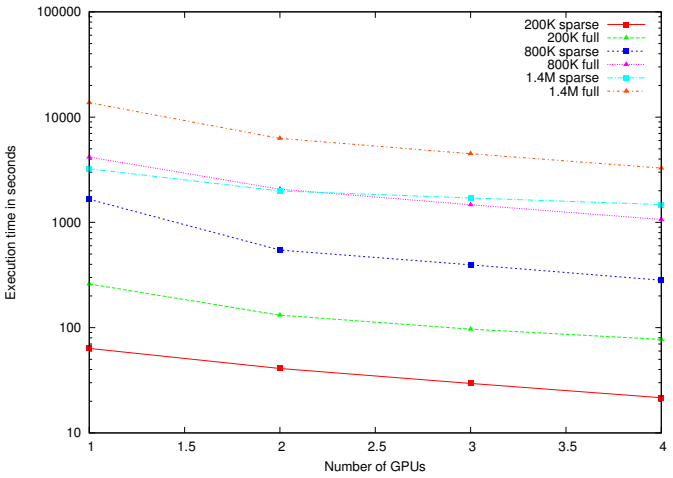

Fig. 8. Execution times to solve sparse and full polynomials of three distinct degrees on multiple GPUs using MPI.

and MPI for the polynomial degree of 200K. For the degree of 800K, the MPI version is a little bit slower than the OpenMP version but for the degree of 1,4 million, there is a slight advantage for the MPI version. In Figure 6, we can see that when it comes to full polynomials, both approaches are almost equivalent.

C. Solving sparse and full polynomials of the same degree on multiple GPUs

In this experiment we compare the execution times of the EA algorithm according to the number of GPUs to solve sparse and full polynomials on multiple GPUs using OpenMP or MPI approaches. We chose three sparse and three full polynomials of degrees 200,000, 800,000 and 1,400,000. Figures 7 and 8 show the execution times to solve sparse and full polynomials of the same degrees with the CUDA-OpenMP version and the CUDA-MPI version, respectively.

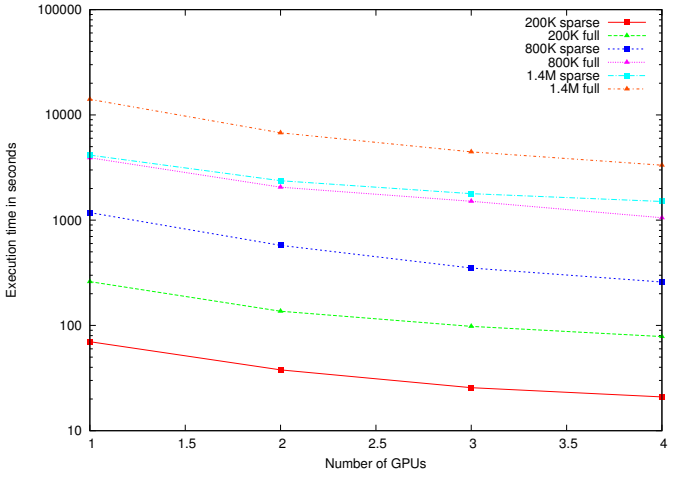

Fig. 7. Execution times to solve sparse and full polynomials of three distinct degrees on multiple GPUs using OpenMP.

In Figure 7 the execution times of the CUDA-OpenMP version to solve sparse polynomials are very low compared to those to solve full polynomials. With sparse polynomials the number of monomials is reduced, consequently the number of operations is reduced and the execution time decreases. Figure 8 shows the impact of sparsity on the efficiency of the CUDA-MPI approach. We can see that the impact follows the same pattern, a difference in execution times in favor of the sparse polynomials.

D. Scalability of the EA method on multiple GPUs to solve very high degree polynomials

These experiments report the execution times of the EA method for sparse and full polynomials of high degrees ranging from 1,000,000 to 5,000,000. In Figure 9 we can see that both approaches (CUDA-OpenMP and CUDA-MPI) are scalable and can solve very high degree polynomials. In addition, with full polynomial as well as sparse ones, both approaches give very similar results.

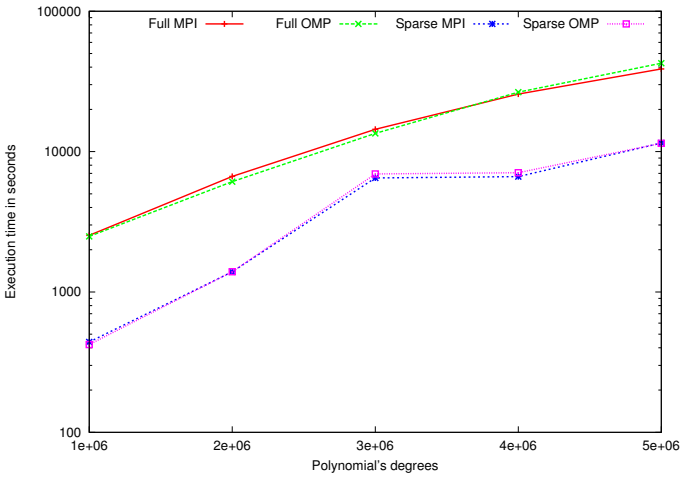

Fig. 9. Execution times in seconds of the Ehrlich-Aberth method to solve sparse and full polynomials of high degree on 4 GPUs for degrees ranging from 1M to 5M

VI. Conclusion

In this paper, we have presented parallel implementations of the Ehrlich-Aberth algorithm to solve full and sparse polynomials, on a single GPU with CUDA and on multiple GPUs using two parallel paradigms: shared memory with OpenMP and distributed memory with MPI. These architectures were addressed by a CUDA-OpenMP approach and CUDA-MPI approach, respectively. Experiments show that, using parallel programming model like OpenMP or MPI, we can efficiently manage multiple graphics cards to solve the same problem and accelerate the parallel execution with 4 GPUs and solve a polynomial of degree up-to 5,000,000 four times faster than on a single GPU.

Our next objective is to extend the model presented here to clusters of GPU nodes, with a three-level scheme: inter-node communications via MPI processes (distributed memory), management of multi-GPU nodes by OpenMP threads (shared memory). Actual platforms may probably also contain purely multi-core nodes without any GPU. This heterogeneous setting may lead to the integration of load balancing algorithms so as to allow an optimal use of hardware resources.

Acknowledgment

This paper is partially funded by the Labex ACTION program (contract ANR-11-LABX-01-01) and the Franche-Comte regional council. Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté. We also would like to thank Nvidia for hardware donation under CUDA Research Center 2016.

References

[1] E. Durand, *Solutions numériques des équations algébriques. Tome I: Équations du type $F(x) = 0$; racines d'un polynôme*. Masson, Paris, 1960.

[2] I. O. Kerner, "Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen. (German) [A complete step method for the computation of zeros of polynomials]," *Numerische Mathematik*, vol. 8, no. 3, pp. 290–294, May 1966.

[3] L. W. Ehrlich, "A modified newton method for polynomials," *Commun. ACM*, vol. 10, no. 2, pp. 107–108, 1967. [Online]. Available: http://doi.acm.org/10.1145/363067.363115

[4] O. Aberth, "Iteration methods for finding all zeros of a polynomial simultaneously," *Mathematics of Computation*, vol. 27, no. 122, pp. 339–344, 1973.

[5] T. L. Freeman, "Calculating polynomial zeros on a local memory parallel computer," *Parallel Computing*, vol. 12, no. 3, pp. 351–358, 1989. [Online]. Available: http://dx.doi.org/10.1016/0167-8191(89)90093-8

[6] G. Loizou, "Higher-order iteration functions for simultaneously approximating polynomial zeros," *Intern. J. Computer Math*, vol. 14, no. 1, pp. 45–58, 1983.

[7] T. Freeman and R. Brankin, "Asynchronous polynomial zero-finding algorithms," *Parallel Computing*, vol. 17, pp. 673–681, 1990.

[8] D. Bini, "Numerical computation of polynomial zeros by means of aberth's method," *Numerical Algorithms*, vol. 13, no. 2, pp. 179–200, 1996. [Online]. Available: http://dx.doi.org/10.1007/BF02207694

[9] R. Couturier and F. Spies, "Extraction de racines dans des polynômes creux de degré élevé," *RSRCP (Réseaux et Systèmes Répartis, Calculateurs Parallèles), Numéro thématique : Algorithmes itératifs parallèles et distribués*, vol. 13, no. 1, pp. 67–81, 2001.

[10] R. Couturier, P. Canalda, and F. Spies, "Iterative algorithms on heterogeneous network computing: Parallel polynomial root extracting," in *High Performance Computing – (9th HiPC'02), Proceedings 9th International Conference*, ser. Lecture Notes in Computer Science (LNCS), S. Sahni, V. K. Prasanna, and U. Shukla, Eds., vol. 2552. Bangalore, India: Springer-Verlag (New York), Dec. 2002, pp. 283–291.

[11] *CUDA C programming guide*. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[12] K. Ghidouche, R. Couturier, and A. Sider, "Parallel implementation of the Durand-Kerner algorithm for polynomial root-finding on GPU," *IEEE. Conf. on advanced Networking, Distributed Systems and Applications*, pp. 53–57, 2014.

[13] "OpenMP application program interface," July 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[14] *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[15] H. Guggenheimer, "Initial approximations in Durand–Kerner's root finding method," *BIT*, vol. 26, no. 4, pp. 537–539, Dec. 1986.

[16] I. Newton, "Tractatus de methodis serierum et fluxionum," in *The Mathematical Papers of Isaac Newton, III*, D. T. Whiteside, Ed. Cambridge University Press, Cambridge, 1670–1671, pp. 32–353.

[17] K. Weierstrass, "Neuer beweis des satzes, dass jede ganze rationale function einer veranderlichen dagestellt werden kann als ein product aus linearen functionen derselben veranderlichen," *Ges. Werke*, vol. 3, pp. 251–269, 1903.

[18] K. Rhofir, F. Spies, and J.-C. Miellou, "Perfectionnements de la méthode asynchrone de Durand-Kerner pour les polynômes complexes," *Calculateurs Parallèles*, vol. 10, no. 4, pp. 449–458, 1998.