Stud. Univ. Babeş-Bolyai Math. 59(2014), No. 4, 557–566

Distributed computing of simultaneous Diophantine approximation problems

Norbert Tihanyi, Attila Kovács and Ádám Szűcs

Abstract. In this paper we present the *Multithreaded Advanced Fast Rational Approximation* algorithm – `MAFRA` – for solving n-dimensional simultaneous Diophantine approximation problems. We show that in some particular applications the Lenstra-Lenstra-Lovász (L^3) algorithm can be substituted by the presented one in order to reduce their practical running time. `MAFRA` was implemented in the following architectures: an Intel Core i5-2450M CPU, an AMD Radeon 7970 GPU card and an Intel cluster with 88 computing nodes.

Mathematics Subject Classification (2010): 68R01, 11J68.

Keywords: Diophantine approximation, rational approximation.

1. Introduction

1.1. Diophantine approximations

Approximating an irrational α with rationals is called Diophantine approximation or rational approximation. The theory of continued fractions provides one of the most effective methods of rational approximation of a real number [1]. *Simple continued fractions* are expressions of the form

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cdots}}$$

where a_i-s are integers with $a_1, a_2 \ldots > 0$. The sequence $C_0 = a_0,\ C_1 = a_0 + \frac{1}{a_1},\ \ldots$ are called *convergents*. Every convergent $C_m = p_m/q_m$ represents a rational number. An infinite continued fraction $[a_0; a_1, \ldots, a_m]$ is called *convergent* if the limit

$$\alpha = \lim_{m\to\infty} C_m$$

This paper was presented at the 10th Joint Conference on Mathematics and Computer Science (MaCS 2014), May 21-25, 2014, Cluj-Napoca, Romania.

exists. It is known that no better rational approximation exists to the irrational number α with smaller denominator than the convergents (see e.g: [2]). Fractions of the form

$$\frac{p_{m-1}+jp_m}{q_{m-1}+jq_m}\ (1 \le j \le a_{m+2}-1)$$

are called *intermediate (or semi-) convergents.* Calculating intermediate convergents can be used to get *every* rational approximation between two consecutive convergents p_m/q_m and p_{m+1}/q_{m+1}. Adolf Hurwitz (1859-1919) proved in 1891 that for each irrational α there are infinitely many pairs (p,q) of integers which satisfy

$$\left|\alpha - \frac{p}{q}\right| < \frac{1}{q^2\sqrt{5}}\,.$$

Approximating more than one irrationals at the same time is called *simultaneous Diophantine approximation.* The challenge in this case is that for given real numbers $\alpha_1, \alpha_2, \ldots, \alpha_n$ and $\varepsilon > 0$ find $p_1, p_2, \ldots, p_n, q \in \mathbb{Z}$ such that

$$\left|\alpha_i - \frac{p_i}{q}\right| < \varepsilon \tag{1.1}$$

for all $1 \le i \le n$. The continued fraction approximation method can be used efficiently for constructing solutions in one or two dimensions. In higher dimensions the situation is more challenging. In 1982 Arjen Lenstra, Hendrik Lenstra and László Lovász invented a polynomial time lattice basis reduction algorithm (L^3) that can be used for solving simultaneous Diophantine approximations [3]. If $\alpha_1, \alpha_2, \ldots, \alpha_n$ are irrationals and $0 < \varepsilon < 1$ then there is a polynomial time algorithm to compute integers $p_1, p_2, \ldots, p_n, q \in \mathbb{Z}$ such that

$$1 \le q \le 2^{n(n+1)/4}\varepsilon^{-n} \text{ and } |q \cdot \alpha_i - p_i| < \varepsilon$$

for all $1 \le i \le n$. The algorithm L^3 can be used effectively for solving Diophantine approximations in higher dimensions, however, it can not be used to generate thousands or millions of $q \in \mathbb{Z}$ that satisfy (1.1) even with varying reduction parameters. Consider the set of irrationals $\Upsilon = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. Let $\varepsilon > 0$ and let us define the set

$$\Lambda(\Upsilon, \varepsilon) = \{k \in \mathbb{N} : \|k\alpha_i\| < \varepsilon \text{ for all } \alpha_i \in \Upsilon\} \tag{1.2}$$

where $\|\cdot\|$ denotes the nearest integer distance function, i.e.

$$\|z\| = \min\{|z-j|, j \in \mathbb{Z}\}\,.$$

In general, the following computational challenges can be stated: (1) generate as many elements of $\Lambda = \Lambda(\Upsilon, \varepsilon)$ as possible in a given time frame, and (2) generate a predefined (huge) number of solutions as fast as possible. In this paper we consider the following number-theoretic challenge:

Challenge: Determine 1 billion elements of the set

$$\Lambda\left(\left\{\frac{\log(p)}{\log(2)}\,, p \text{ prime}\,, 3 \le p \le 31\right\}, 0.01\right) \tag{1.3}$$

as fast as possible. This challenge is a 10-dimensional simultaneous Diophantine approximation problem. Generating such a huge amount of integers with L^3 would be very time-consuming on an average desktop PC. The first two authors of this paper recently presented a method for solving n-dimensional Diophantine approximation problems efficiently [4]. The main idea is the following:

Theorem 1.1. *Let $\Upsilon = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ be a set of irrationals and $\varepsilon > 0$ real. Then there is a set Γ_n with 2^n elements with the following property: if $k \in \Lambda$ then $(k+\gamma) \in \Lambda$ for some $\gamma \in \Gamma_n$.*

It was also presented that the generation of Γ_n can be done very efficiently for small dimension (e.g: $n < 20$). In our particular case using the set Γ_{10} it is possible to generate arbitrarily many integers $k \in \Lambda$.

The main goal of this paper is to improve the implementation of the existing algorithms and develop an even faster method than the one presented in [4]. We refer to this new algorithm as `MAFRA` – *Multithreaded Advanced Fast Rational Approximation.*

1.2. Practical usage of MAFRA

Fast algorithms for solving Diophantine approximations can be used in many fields of computer science. In some particular applications the L^3 algorithm can be substituted by `MAFRA` in order to reduce their practical running time. We used `MAFRA` for locating large values of the Riemann zeta function on the critical line. The Riemann-Siegel formula can be calculated by

$$Z(t) = 2 \sum_{n=1}^{\lfloor\sqrt{t/2\pi}\rfloor} \frac{1}{\sqrt{n}} \cos(\theta(t) - t \cdot \ln n) + O(t^{-1/4}), \tag{1.4}$$

where $\theta(t) = \arg(\Gamma(1/4 + \frac{it}{2})) - \frac{1}{2} t \ln \pi$. In 1989 Andrew M. Odlyzko presented a method for predicting large values of $Z(t)$. "We need to find a t for which there exist integers $m_1, \ldots, m_n$ such that each of $t \ln p_k - 2\pi m_k$ is small $(1 \le k \le n)$" [5]. This is a simultaneous Diophantine approximation problem like (1.3). By applying `MAFRA` one can solve this kind of approximation problem much faster than with L^3 for small dimensions ($n < 20$). We implemented `MAFRA` in order to be able to measure the practical running time in different architectures.

2. Algorithms for solving Diophantine approximation problems

It is known that Algorithm 2.1 solves our challenge efficiently [4].

Algorithm 2.1. – **(FRA)** – **Fast Rational Approximation**

```
Require: bound                                  ▷ default is one billion
Require: $k$                          ▷ starting point, the default is zero
 1: $\Gamma \leftarrow$ Apply Algorithm PRECALC from [4]
 2: $\Upsilon \leftarrow \frac{\log(p)}{\log(2)}$, $p$ prime, $3 \le p \le 31$
 3: counter $\leftarrow 0$, $\varepsilon \leftarrow 0.01$
 4: while counter $<$ bound do
 5:     for $i = 1 \to 1024$ do
 6:         find $\leftarrow$ TRUE
 7:         for $j = 1 \to 10$ do
 8:           $a \leftarrow \textsc{Frac}((k + \Gamma[i]) \cdot \Upsilon[j])$
 9:           if $(a > \varepsilon)$ and $(a < 1 - \varepsilon)$ then
10:               find $\leftarrow$ FALSE
11:               break                                    ▷ Leave the for loop
12:           end if
13:       end for
14:       if find $=$ TRUE then
15:           $k \leftarrow k + \Gamma[i]$
16:           counter $\leftarrow$ counter $+ 1$
17:           break
18:       end if
19:   end for
20: end while
```

The test system was an Intel®Core i5-2450M CPU with Sandy Bridge architecture and the development environment was the PARI/GP computer algebra system. Using this setup it was possible to produce 100 000 appropriate integers within 22.16 seconds. In order to achieve better performance the development environment had been changed to native C using the GNU MP 5.1.3 multi precision library. In Algorithm 2.1 the Precalc function calculates Γ_{10} in few minutes. After the calculation of the 1024 elements of Γ_{10} one can generate arbitrarily many $k \in \Lambda$ very efficiently. With the improved C code it was possible to produce 100 000 integers within 2.65 seconds. This is approximately 10 times faster than the PARI/GP implementation.

It is important to note a significant difference between our 10-dimensional Challenge (1.3) and the 7-dimensional Challenge presented in [4]. In that paper the solution set was defined in the following way:

$$\Omega(\Upsilon, \varepsilon, a, b) = \{k \in \mathbb{N} : a \le k \le b, \|k\alpha_i\| < \varepsilon \text{ for all } \alpha_i \in \Upsilon\}.$$

As it can be seen, the elements of the Ω are bounded. In (1.2) we redefined Ω without boundaries. This "small" change of the definition allows us to design and develop an even faster algorithm.

Algorithm 2.2. – **(AFRA)** – **Advanced Fast Rational Approximation**

```
Require: bound                                   ▷ default is one billion
Require: $k$                              ▷ starting point, the default is zero
 1: $\Gamma \leftarrow$ Apply Algorithm Precalc from [4]
 2: $\Upsilon \leftarrow \frac{\log(p)}{\log(2)}$, $p$ prime, $3 \le p \le 31$
 3: $\varepsilon \leftarrow 0.01$
 4: counter $\leftarrow 0$
 5: while counter $<$ bound do
 6:   sum $\leftarrow 0$
 7:   for $i = 1 \to 10$ do
 8:       $a \leftarrow \textsc{Frac}(k \cdot \Upsilon[i])$
 9:       if $(a < \varepsilon)$ then
10:           sum $\leftarrow$ sum $+ 2^i$
11:       end if
```

```
12:     end for
13:     sum ← $abs$(sum − 1024)                    ▷ binary complementer
14:     counter ← counter + 1
15:     $k \leftarrow k + \Gamma$[sum]
16: end while
```

Algorithm 2.2 (AFRA) is substantially different from Algorithm 2.1 (FRA). Let $k \in \Lambda$. FRA always finds the smallest $\gamma \in \Gamma_{10}$ where $(k+\gamma) \in \Lambda$. It is easy to see that in the worst case this algorithm goes through all the 1024 elements of Γ_{10} (see Algorithm 2.1, line 5). In each step the algorithm has to check whether $(k+\gamma) \in \Lambda$ or not (see line 9). AFRA finds one element from Γ_{10} — not necessary the smallest one[1] — that satisfies $(k+\gamma) \in \Lambda$ (Lemma 8 in [4] ensures finding the appropriate $\gamma \in \Gamma_{10}$ efficiently). Algorithm 2.2 is therefore faster, however, adding some γ to k produces larger values in Λ. It can be concluded that FRA is a better choice for solving bounded challenges like $\Omega(\Upsilon, \varepsilon, a, b)$. For solving unbounded challenges, like our particular 10-dimensional case, AFRA is much better. We implemented Algorithm 2.2 in native C. In our test system we were able to generate 100 000 integers $\in \Lambda$ in 0.434 seconds[2]. This is almost ten times faster than the Algorithm 2.1 implementation.

Let us compare the algorithms FRA and AFRA with exact numbers. Consider the following challenge: generate as many integers as possible in the set

$$\Omega\left(\left\{\frac{\log(p)}{\log(2)}\,,p \text{ prime}\,, 3 \le p \le 31\right\}, 0.01, 0, 2\times 10^{19}\right) \tag{2.1}$$

This challenge differs from (1.3) since the elements of Ω are bounded. As we mentioned FRA is a better choice for a bounded challenge. Solving (2.1) by FRA one can produce 13 different integers between 0 and 2×10^{19}. These integers are presented in Table 1. It is easy to verify that every integer k in Table 2 satisfies the following:

$$\left\| k\frac{\log(p)}{\log(2)} \right\| < 0.01$$

for all $3 \le p \le 31$.

Table 1. FRA output between 0 and 2×10^{19}

1023317259883927888	4791256480457711184	7100801081230345000
1711993379226146170	2088787301283524566	3423106890630466630
5441342799508541730	7540063840126351339	8406797017385611672
10118790396611757842	10503998465875331568	11021951848184774212
19036050657750584878		

These integers were generated in 0.015 seconds. AFRA is almost 10 times faster than FRA, however, inappropriate for solving this particular "bounded" challenge.

[1]The set of integers in Γ_{10} are ordered in the following way: every integer in Γ_n is represented by an n-dimensional binary vector (see Lemma 8 in [4]). Γ_{10} contains integers ordered by the values of this binary vector (e.g: 0000000000, 0000000001, 0000000010, 0000000011 etc.)

[2]During the measurements Input/Output costs are not cummulated. Displaying the 100 000 integers from the memory would take approximately 5-6 seconds.

With `AFRA` we can produce only one integer solution, which is 2298677471355273619. The next integer would be 18396312148683633119 6 which is already out of the upper bound 2×10^{19}.

We conclude that challenge (1.3) is unbounded, so when the size of the integers is unimportant then Algorithm `AFRA` is the right solution.

3. Computing methods and results

To make the generation even faster we modified our C code in order to be able to run in parallel using pthreads (IEEE Std. 1003.1c-1995.). We refer to the multithreaded version of `AFRA` as `MAFRA`– *Multithreaded Advanced Fast Rational Approximation* algorithm.

In this section we present the measured running time of `MAFRA` for different architectures. The first test environment was a simple Sandy Bridge Intel Core i5-2450M with 4 GB RAM. The second hardware was a Super Computing Cluster called ATLAS with 90x Intel Xeon E5520 Nehalem Quad Core 2.26 GHz Processors and 0.6TB RAM. The third hardware was an ATI Radeon 7970 GPU card.

3.1. Test – Core i5-2450M Laptop

Our first test environment was a simple home desktop PC. It was an Intel Core i5-2450M Sandy Bridge CPU with 4 GB RAM having 2 cores. Generating 100 000 integers $\in \Lambda$ for solving the 10 dimensional challenge with the algorithm `MAFRA` took 0.234 sec. Our newly implemented, optimized and multithreaded C code is effective, however, generating 1 billion elements of (1.3) with this architecture would take approximately 39 minutes.

3.2. Test – ATLAS Computing Cluster

Our second test environment was the ATLAS Supercomputing Cluster that is operating in the Eötvös Loránd University, Budapest. The most important characteristics of ATLAS are the following: the architecture consists of one dedicated Headnode and 44 Computing nodes.

1x Headnode:

1. 2x Intel Xeon E5520 Nehalem Quad Core 2.26 GHz Processor with 8 MB cache (HyperThreading OFF)
2. 72 Gbyte RAM
3. 10 Gbit eth interface to the 44 computing nodes

44x Computing Nodes:

1. 2x Intel Xeon E5520 Nehalem Quad Core 2.26 GHz Processor with 8 MB cache (HyperThreading ON)
2. 12 Gbyte RAM

Each Nehalem Quad core CPU has 4 physical cores with SSE extension. Each node has a 2×36.256 GFLOP/sec peak performance (see [6]) calculated by the following formula:

$FLOPS = 4\,\text{cores} \times 2.266\text{GHz} \times 2\,(\text{SIMD double prec.}) \times 2\,(\text{MUL, ADD})$
$= 36.256$ GFLOP/sec.

There are 44 computing nodes which contain 88 physical CPU. The total number of physical cores are 352 (4×88). With hyper-threading the number of cores can be doubled to 704 virtual core. The peak performance of the ATLAS Computing Cluster is $72.512 \times 44 = 3190.528$ GFLOP/sec. With full performance ATLAS takes 12.6 kW, 34.2 A, and cosFI= 0.95.

Generating 100 000 integers in one computing node took approximately 0.175 sec. Remember that in the previous test the Intel Core i5-2450 had 2 cores with 4 threads. If the number of threads is less than the number of dimensions then the multithreaded running is obvious; every thread checks whether $(k+\gamma)\cdot\Upsilon[i] < \varepsilon$ for all $i < n$ where n denotes the dimension. ATLAS has 44 different nodes which are much more than the number of dimensions in our particular case. If one wanted to use all of the cores then the best way would be to run 44 copies of AFRA in each node. In this case each node should start from different starting points. Generating 44 different appropriate starting points for each copies of AFRA can be done very effectively with the L^3 algorithm. Let $\alpha_1, \alpha_2, \dots, \alpha_n$ be irrational numbers and let us approximate them with rationals admitting an $\varepsilon > 0$ error. Let $X = \beta^{n(n+1)/4}\varepsilon^{-n}$ and let the matrix A be the following:

$$A = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha_1 X & X & 0 & \dots & 0 \\ \alpha_2 X & 0 & X & \dots & 0 \\ \vdots & & & & \vdots \\ \alpha_n X & 0 & 0 & \dots & X \end{bmatrix}.$$

Applying the L^3 algorithm for A the first column of the resulting matrix contains the vector $[q, p_1, p_2, p_3, \dots, p_n]^T$ which satisfies

$$\left|\alpha_i - \frac{p_i}{q}\right| < \frac{\varepsilon}{q} \text{ and } 0 < q \le \beta^{n(n+1)/4}\varepsilon^{-n}$$

for all $1 \le i \le n$, where β is an appropriate reduction parameter. Using `MAFRA` in accordance with L^3 it is possible to generate 4.4 millions of integers within $0.175 + \delta$ seconds where δ is the generating time of the 44 starting points not exceeding 5000 ms. With the ATLAS Computing Cluster calculating exactly one billion integers that satisfy (1.3) took approximately 39.7 seconds.

Generating the 44 integers as starting points with L^3 can be done very effectively, however, we would like to emphasize again that the L^3 algorithm is ineffective in generating many solutions (e.g. one billion).

3.3. Test – ATI Radeon 7970 GPU

The third test environment was a Sapphire Vapor-X ATI Radeon 7970 6GB GDDR5 GHz Edition GPU card. Modern graphic cards can be other promising solutions for solving high performance computations. Clearly, in order to implement another fast method for our Diophantine approximation problem one has to take into consideration the usage of GPU cards. In our case the multithreaded version of `FRA` and `AFRA` were implemented for the GPU. In the first step, however, we faced with the following problem: there were not any fast *quadruple* precision packages on the GPU. Although some similar packages for the older GPU cards were found written by Andrew Thall [7] and Eric Bainville [8], these packages found to be inappropriate to solve our particular challenge. The problem with the package written by Andrew Thall is that it uses too much branching and function calling in the program which costs a lot clock cycles. It comes from the behaviour of the graphical processing unit which evaluates both the `if` and the `else` part of the conditional, and after the computation it uses that data where the logical value was TRUE. `FRA` and `AFRA` contain a lot of logical evaluation, so the usage of this package was not convenient for our purposes. The other package, which was written by Eric Bainville, is faster, but it is for fixed point numbers which was inappropriate, as well.

In conclusion, we developed our own multiplication, addition, subtraction and truncation methods. We applied the Karatsuba multiplication algorithm and some bitwise tricks for the addition and truncation methods. After measuring the running speeds on this architecture it turned out that using the `AFRA` algorithm on the GPU approximately 50 times performance drop-down could be measured without using the L^3 algorithm for the generation of the starting points[3].

`bound`	AFRA Running speed in seconds
1	0.0254221
10	0.183748
100	1.35351
1000	12.7255
10000	127.038
100000	1200.7

After examining `AFRA` we can state that the main problem with this "linear" algorithm is that it was not possible to distribute enough threads on the GPU. Consider for example our 10-dimensional case. One had to add the 1024 integers to the partial results and then multiply them with the irrationals. The problem with this solution is that in the quadruple–adder kernel it was not possible to send in enough threads lowering or hiding the latency. In our case the global work size was twice as big as the local work size, which leaded to performance drop-down. In order to avoid the big performance drop-down we utilized every threads on the GPU just like in the ATLAS Super Cluster. For example, if we want to use 2048 threads on the GPU, then we would have to generate 2048 different starting points with the L^3 algorithm to feed all the threads on the GPU. We also modified a bit the number representation in order to achieve higher speed on this architecture. In that particular case our measurements show that generating 100 000 different integers on the 7970 GPU it is 4 times faster than on the CPU.

[3]Measuring speeds on the GPU is only an approximation.

Combination of the CPU version of L^3 and the GPU version of `MAFRA` turned out to be a very effective way to solve simultaneous Diophantine approximation problems. A supercomputer with GPU accelerators would be a nice solution for this problem.

4. Further Researches

As we stated in the introduction we used `MAFRA` for locating large values of the Riemann zeta function on the critical line. It was possible to substitute L^3 with `MAFRA` in order to achieve a much better performance of finding large values. We have implemented `MAFRA` algorithm to the GRID system of the Hungarian Academy of Sciences and solving simultaneous Diophantine approximation problems very effectively. We plan to continue our research in this direction.

Acknowledgment. The authors gratefully acknowledge the constructive comments of the referee. The authors would like to thank the opportunity to grant access to the ATLAS Super Cluster operating at Eötvös Loránd University. The research of the second author was partially supported by the European Union and co-financed by the European Social Fund (ELTE Támop-4.2.2/B-10/1-2010-0030).

References

[1] Khinchin, A.Y., *Continued fractions*, Translated from the third (1961) Russian edition, Reprint of the 1964 translation, Dover, Mineola, NY, 1997.

[2] Niven, I., Zuckerman, H.S., Montgomery, H.L., *An Introduction to the Theory of Numbers*, Fifth Edition, 1991, Chapter 7, pp. 338.

[3] Lenstra, A.K., Lenstra Jr., H.W., Lovász, L., *Factoring polynomials with rational coefficients*, Mathematische Annalen, **261**(1982), no. 4, 515–534.

[4] Kovács, A., Tihanyi, N., *Efficient computing of n-dimensional siultaneous Diophantine approximation problems*, Acta Univ. Sapientia, Informatica, **5**(2013), no. 1, 16–34.

[5] Odlyzko, A.M., *The 10^{20}-th zero of the Riemann zeta function and 175 million of its neighbors*, 1992
http://www.dtc.umn.edu/∼odlyzko/unpublished/index.html

[6] *Peak performance of Intel CPU's*,
http://download.intel.com/support/processors/xeon/sb/xeon_5500.pdf

[7] Thall, A., *Extended-Precision Floating-Point Numbers for GPU Computation*,
http://www.caesar.elte.hu/hpc/atlasz-hw.html

[8] Web page of Eric Bainville, http://www.bealto.com/cv.html

[9] PARI/GP Computeralgebra system, http://pari.math.u-bordeaux.fr/

[10] Web page of the ATLAS Computing Cluster, Eötvös Loránd University,
http://www.caesar.elte.hu/hpc/atlasz-hw.html (in hungarian).

[11] Thomadakis, M.E., *The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms http://sc.tamu.edu/systems/eos/nehalem.pdf*, Texas *A&M* University, March 17, 2011.

[12] Sapphire Vapor-X HD 7970 GHz Edition 6GB DDR5,
http://www.sapphiretech.com/presentation/product/

Norbert Tihanyi
Eötvös Loránd University, Faculty of Informatics
Department of Computer Algebra
Budapest, Hungary
e-mail: `ntihanyi@compalg.inf.elte.hu`

Attila Kovács
Eötvös Loránd University, Faculty of Informatics
Department of Computer Algebra
Budapest, Hungary
e-mail: `attila.kovacs@compalg.inf.elte.hu`

Ádám Szűcs
Eötvös Loránd University, Faculty of Informatics
Department of Computer Algebra
Budapest, Hungary
e-mail: `adam.szucs@compalg.inf.elte.hu`