Received 27 July 2020; revised 13 November 2020; accepted 26 November 2020. Date of publication 9 December 2020;
date of current version 15 January 2021.

Digital Object Identifier 10.1109/OJAP.2020.3043541

Large Minimum Redundancy Linear Arrays: Systematic Search of Perfect and Optimal Rulers Exploiting Parallel Processing

FABIAN SCHWARTAU [1], YANNIC SCHRÖDER [2], LARS WOLF [2] (Senior Member, IEEE), AND JOERG SCHOEBEL [1]

[1] Institut für Hochfrequenztechnik, Technische Universität Braunschweig, 38106 Braunschweig, Germany

[2] Institute of Operating Systems and Computer Networks, Technische Universität Braunschweig, 38106 Braunschweig, Germany

CORRESPONDING AUTHOR: F. SCHWARTAU (e-mail: fabian.schwartau@ihf.tu-bs.de)

This work was supported in part by the German Research Foundation and in part by the Open Access Publication Funds of Technische Universität Braunschweig.

ABSTRACT Minimum Redundancy Linear Arrays (MRLAs) are special linear arrays that provide the narrowest main lobe in the radiation pattern possible for a given number of antennas. We found that the calculation of MRLAs is the same as for the mathematical problem of perfect sparse rulers. Finding perfect rulers (or MRLAs) is a hard problem, as there is no proven mathematical rule to design them. They can only be found by constructing ruler candidates via an exhaustive search while ensuring that no ruler with less redundancy exists. We revisited the problem of sparse ruler construction and used two exhaustive search algorithms to compute longer rulers than previously published. Further, we present an approach to accelerate the execution by distributing the recursive search algorithms over multiple computers. Our compute cluster found perfect rulers with all lengths up to 244 in 443 years of combined CPU time. All found rulers are provided to the research community. Additionally, we confirm previously known Low Redundancy Linear Arrays being MRLAs. Our results show that larger perfect rulers do not always require equal or more marks (antennas) but can sometimes be constructed with fewer marks than the previous ruler.

INDEX TERMS Aperture synthesis, linear antenna arrays, redundancy, thinned array, minimum redundancy linear array, MRLA, sparse ruler, perfect ruler, optimal ruler.

I. INTRODUCTION

MINIMUM Redundancy Linear Arrays (MRLAs) are a well-known type of linear antenna arrays, which provide maximum angular resolution for a given number of antennas [1]. An Minimum Redundancy Linear Array (MRLA) is constructed by placing the individual antennas on discrete spots so that there is a minimum of redundant spacings between any of the elements. The calculation of such MRLAs is becoming extremely time-consuming for large arrays. Computational complexity increases drastically with increasing MRLA length as they can only be found via exhaustive search. Available computing resources (mainly CPU time) are the limiting factor to find longer MRLAs. Previously, arrays with a length of up to 113 have been found [2]. For a given array length or number of antennas, there is a finite number of possible arrays fulfilling the criteria of an MRLA. The goal of an exhaustive search is to find all possible results.

Recent research focused on finding Low Redundancy Linear Arrays (LRLAs) as a pattern is observable in known MRLAs that can be exploited to construct large arrays with low redundancy without investing much CPU time. However, these arrays cannot be proven to be MRLAs without an exhaustive search for arrays with lower redundancy.

It turns out that the mathematics behind the calculation of an MRLA is identical to *perfect sparse rulers* and similar to the graceful graph problem, which are well-known problems in the mathematical community. In comparison to MRLA,

the search for sparse rulers was conducted up to a length of 213 [3], [4].

The results shown in [3] are probably taken from [4], as they reach the same length, and the program presented in [3] is, according to our tests, not fast enough to produce such large rulers. This would mean that the results with a length from 114 up to 213 have not been verified independently. We were able to verify these results by using the alternative Blanton&McClellan algorithm [2]. The algorithm was implemented in *C*, optimized for speed, and adopted to allow parallel execution on multiple processor cores and computers. Additionally, we modified the algorithm presented in [4] to be compatible with our parallel computation back-end and used it to compute sparse rulers (i.e., MRLAs) up to a length of 244. All our results, including the source code, is provided in [5].

The following Section II introduces some definitions of rulers that will be used throughout this article. It describes the different type of sparse rulers and the relation to MRLAs. Section III describes the two used algorithms and their differences, followed by the description of the parallelization approach we took in Section IV. The software for the distribution is explained in Section V. Finally, the results and a conclusion are given in Sections VI and VII, respectively.

The contributions of this work include the finding that MRLAs and *perfect* sparse rulers are identical problems, software implementations of two algorithms for the search of MRLAs or *perfect* sparse rulers, the parallelization of search across multiple computers, the independent confirmation of rulers between length 114 and 213, and additional results up to a length of 244.

II. RULER PROPERTIES

Peter Luschny gives definitions for different properties of rulers [6]. Three parameters define every ruler: The number of tick marks M on the ruler (i.e., antennas), its length L, which is the distance between the two outermost markings (the maximum length the ruler can measure), and a list, that indicates where on the ruler the markings are. Note that all markings are placed on a fixed grid (like an array), so distances between marks are without unit and always an integer value. The definitions are as follows:

a) *Fully Populated Ruler:* A *fully populated* ruler of length L contains $M = L+1$ marks. This is basically the same as an office desk ruler with a mark for every Centimeter or Inch. With such a ruler, it is trivial to find two marks with any distance between them in the range 1 to L. However, such a ruler contains a lot of redundancy, as there are multiple different pairs of marks that are the same distance apart.

b) *Sparse Ruler:* A *sparse* ruler of length L contains $M < L + 1$ marks. This removes redundant distances between marks. However, a sparse ruler does not necessarily contain all distances from 1 to L.

c) *Complete Ruler:* A *complete* ruler is any ruler that contains all distances from 1 to L without specifying the number of marks M. Every *fully populated* ruler is a complete ruler.

d) *Perfect Ruler:* A *perfect* ruler is a *sparse* and *complete* ruler, that has the minimum number of marks M for a given length L. No complete ruler with less marks can be found for the given length L. However, there can be multiple *perfect* rulers for any L. A *perfect* ruler is what we call an MRLA.

e) *Optimal Ruler:* An *optimal* ruler is a *perfect* ruler with the maximum length L for a given number of marks M. No longer ruler with the same number of marks can be constructed. If an MRLA for an antenna array is needed, where the number of antennas is already known, an *optimal* ruler with M antennas (marks) will be the longest antenna array that can be constructed.

Other research provides the concept of LRLAs [7]–[11]. These arrays offer a lower redundancy than fully populated arrays while not guaranteeing the perfect solution with minimum redundancy, i.e., it is not necessarily a *perfect* ruler.

III. THE ALGORITHMS

Two different algorithms are used to compute *complete* rulers in this work. The first algorithm was introduced by Blanton & McClellan [2] and is named *Blanton&McClellan algorithm* in the rest of this article. The other algorithm was introduced by Arch D. Robison [4] and is thus named *Robison algorithm*. Both deterministically yield identical results and are based on the same principle: Their inputs are the desired length L of the array and the number of marks (i.e., antennas) M to be placed (M is called N by Blanton & McClellan). They yield either zero results if no *complete* ruler with M marks could be found or the list of *complete* rulers with M antennas. Note that the algorithms do not compute *perfect* rulers. Found rulers are *perfect* rulers if the algorithm was unable to find results with any smaller value for M. The general approach to find *perfect* rulers is to choose a length L and start with the lowest possible M, which will be defined below. If the algorithm cannot find any results for this combination, M has to be incremented. This process is repeated until at least one result is found, which is, by definition, a *perfect* ruler.

Both algorithms used for ou search start with an empty ruler except for marks at the two ends. Then, zero, one, or two marks are placed at certain positions resulting in up to four more populated rulers. Afterward, the function calls itself recursively for the new rulers, creating a tree of function calls (i.e., rulers). The recursive function first checks the current state of the ruler. If all marks are placed, and all distances occur in the ruler, it is a complete ruler, which is saved. Additionally, the function does a few checks, if the goal of finding a complete ruler can still be reached. For example, if the number of free spaces in the ruler is less than the number of marks left to be placed, there is no way this recursion path can lead to a ruler with M marks, and the function returns immediately. These checks cut off branches of the tree to be searched, which significantly reduces the execution time. The differences between the two algorithms are within the position where new marks are placed and the checks that are done to increase speed. There is a not yet

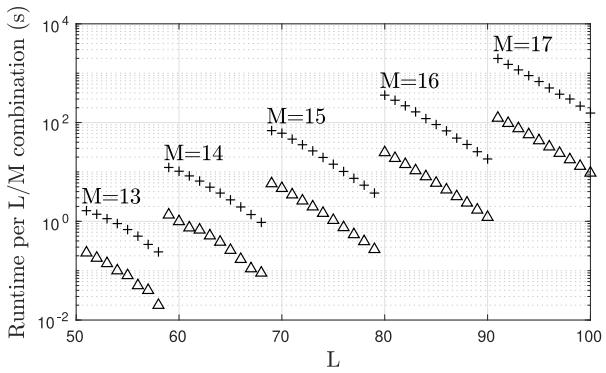

FIGURE 1. CPU time for each combination of *L* and *M* calculating all rulers with the Blanton&McClellan algorithm (+) and the Robison algorithm (△).

complete ruler at every stage of both algorithms, which is to be extended with new marks. We call these intermediate rulers *templates*.

The algorithms' execution can also be aborted if the template would not yield all distances up to L, even if every additional mark would yield the maximum number of new distances. This condition can be computed via Eq. (12) from [2]. By rearranging this formula, the minimum number of antennas M_{min} for any given template can be computed:

$$M_{min} = \left\lceil \frac{1}{2} + \sqrt{\frac{1}{4} + 2L - 2k + P(P-1)} \right\rceil. \qquad (1)$$

k is the number of distances already present in the template, P is the number of already placed marks, and $\lceil\cdot\rceil$ is the ceiling function.

We implemented the Blanton&McClellan algorithm in the *C* programming language as a standalone application based on the explanations in the original paper. Like the original version, it needs values for L and M as input. However, our version can be supplied with a *template*. This allows starting the computation not only with an empty ruler but deep in the recursion tree. It is then possible to start a parallel execution at different points of the tree.

For the Robison algorithm, we used the author's code, which is written in *C++*, and modified it to fit the interface of the other algorithm. The Robison algorithm is designed to exploit modern x86 processors' capabilities by using Streaming SIMD Extensions (SSE) instructions on 128-bit registers. This improves execution speed significantly. Fig. 1 compares the execution times of both implementations.

IV. PARALLELIZATION OF THE ALGORITHMS

There are several ways to implement distributed calculation of spare rulers. The simplest may be to distribute multiple combinations of L and M over multiple computers or processes, where each process calculates one combination. However, to run hundreds or even thousands of parallel processes, it is impossible to provide enough L/M combinations. Further, computing power on a certain combination will be wasted if another process found a solution for a smaller M, and thus all solutions of the current process cannot be *perfect* rulers. If the task was only parallelized via different L/M combinations, certain computations would take a very long time. For example, computing that there is no *perfect* ruler of length 195 with 24 marks takes 697 days of CPU time on a single processor, using the Blanton&McClellan algorithm. Any power outage would result in a loss of the already invested calculation time, as there would be no intermediate results to continue from.

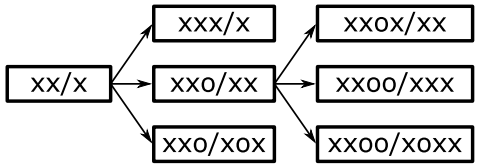

FIGURE 2. Tree structure of MRLA templates. Each template can be split in multiple longer templates. Compare with [2].

A better approach is to execute the recursive algorithm down to a certain level beforehand and interrupt the calculation. This way, we can build a tree of templates, as shown in Fig. 2 for the Blanton&McClellan algorithm. In the representation of a template, as shown in Fig. 2, an x represents a mark (or antenna) and an o represents an empty place. The slash splits the template into its left and right parts. The resulting templates can now be distributed to multiple processes, and the remaining recursions can execute in parallel. The list of templates is the same regardless of L and M, as long as each template is short enough and does not contain more than M marks. It is now possible to define a break level parameter d, which defines how many recursions the algorithm has executed before starting the parallel computation. Two different approaches to determine the parameter d are introduced in the following sections.

A. FIXED BREAK LEVEL

The fixed break level strategy was implemented and tested with the Blanton&McClellan algorithm. It can also be adopted for the Robison algorithm. In this case, a break level of $d = 2$ will result in the default template `xx/x` as two distances are already placed. The higher the break level, the more templates will be generated. A break level of 7 will result in 197 templates, while $d = 12$ will create 30228 templates. The number of templates rises exponentially with a base of roughly 3.

It turned out that the algorithm's execution time differs vastly for templates of the same break level. Fig. 3 (solid line) shows the execution time for the set of templates for $L = 113$ and $M = 19$ with fixed break level of $d = 9$. Many jobs finish within a very short time, while some jobs take very long. 90 % of the jobs are finished within 155 seconds, while the longest takes more than 40 minutes. It has to be pointed out that all jobs for Fig. 3 were calculated on the same machine. This eliminates runtime differences due to different hardware. The observed heterogeneity of runtimes is problematic, as workers that computed short jobs might be idle when waiting for other workers to finish long jobs before

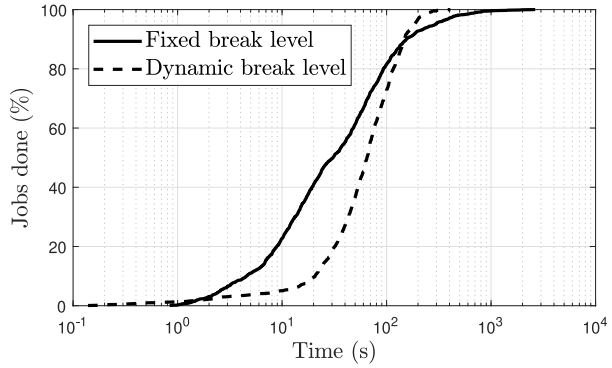

FIGURE 3. Cumulative distribution function of the calculation time with $L = 113$ and $M = 19$ using fixed (solid line) or dynamic (dashed line) break level. 1489 ($d = 9$) and 1554 (maximum runtime 17 s) templates, respectively.

the next L/M combination can start. Additionally, very short runtimes should be avoided, as the communication overhead between the central server and workers would become the dominating factor.

Some templates tend to have short runtimes as their precomputed pattern of antennas cannot generate rulers, and thus computation is aborted early. The long-running templates have patterns that are likely to create a ruler. Thus, the algorithm has to take many recursions before it can determine if a new ruler is found. One way to get around the problem of extremely different runtimes is to make sure that a template that usually requires more time is started very early. This way, there is a lower chance of having a job started at the very end, requiring much more time than the others to finish. Nevertheless, experimental results show that the runtime difference is too much for larger L to be fully compensated by this approach. Further, this will not solve the communication overhead problem.

B. DYNAMIC BREAK LEVEL

Another approach is to treat each template individually. By default, a template is initialized with a certain L, while M is calculated from (1). Once the calculation is finished, the template is handled according to the flow graph in Fig. 4. If the template yielded one or more rulers, all other templates for higher M of the same L are canceled. Those other templates would likely find rulers as well, but they cannot be *perfect*. Next, if the computation duration was larger than the target split time, the template is split into its sub-templates. This limits the execution time for subsequent runs of the algorithm. Lastly, the template(s) are added to the database. If any template for the same L with smaller M yielded a ruler, the value of L is incremented by one, and the value of M is reset to M_{min} as computed via Eq. (1). If no other template found a ruler for a smaller M, the value of L is kept, and M is incremented instead.

Note that the fixed break level has the advantage that a complete set of deeply split rulers can be generated for any L/M combination right away. On the other hand, the dynamic

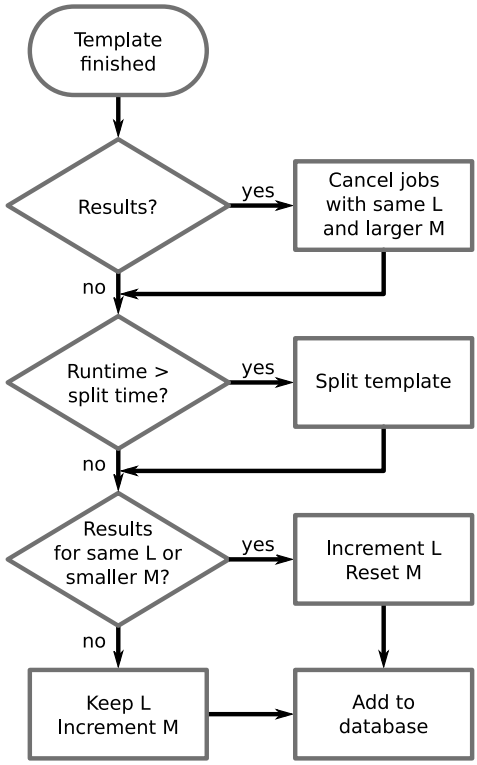

FIGURE 4. Flow graph for a finished template. The operations are executed by the housekeeper at regular intervals for all finished templates.

break level depends on other combinations to be calculated beforehand, which generated the split templates.

The approach of dynamic break levels solves the issue of largely different runtimes as the templates are split dynamically, whenever needed. If a template's runtime was more than a predefined threshold, it will be split into its sub-templates for the next iteration that will run in parallel and thus require less time each. This way, all the templates should require roughly the same time, regardless of L and M. Fig. 3 shows the distribution of the required time for each template using dynamic break levels with $L = 113$ and $M = 19$ (dashed line). While the fixed break level leads to a job requiring more than 40 minutes, this value was reduced to less than 7 minutes using dynamic break levels. The overall heterogeneity of the runtimes is reduced significantly. The split time, i.e., the maximum time before a template is split, is set to 17 s, which yields roughly the same number of templates as for the fixed break level.

Although templates were split when their execution time exceeded 17 s, most templates required more than this time, see Fig. 3. $L = 113$ requires $M = 19$ antennas, while the templates were split after computing $L = 112$ which requires $M = 18$ antennas. The increase of M increases the complexity and, therefore, the execution time by more than an order of magnitude. This also shows the limitation of this approach: The templates can only be split based on past computations. Whenever the number of antennas needs to be increased, the templates' runtimes will exceed the target time.

V. JOB DISTRIBUTION SYSTEM

To manage and distribute the jobs to be computed, a central server is required. A worker (client), running on each participating computer, registers itself with the server and requests jobs to run. Each job is then passed to the calculation programs, implementing either the *Blanton&McClellan* or the *Robison* algorithm, depending on the configuration of the server. The worker reports the results back to the server once they are finished. The server can inform the workers to cancel a specific job if it turns out that the results of that job are not required anymore. Both the server and the worker are written in Python 3. The server is separated into two programs. One provides the API for the workers, e.g., distributing the jobs or receiving the results and saving them in a MySQL database. It also generates a dashboard website, which gives an overview of the current state, the number of rulers already found for a combination of L/M, and other information. For this purpose it utilizes the Bottle.py[1] HTTP-Server. The second part of the server is called the *housekeeper*. It takes care of the database, e.g., splitting templates with high runtimes, calculating the new L and M, or marking duplicate (mirrored) results.

The software supports a special mode called *stop on result* where all templates are canceled/skipped, and L is increased once a single result from any template with a lower or the same M is known. At that point, the required number of marks for this L and at least one *perfect* ruler is known. The algorithm then continues with the next L. This way, it is possible to find the L/M combinations for all *perfect* rulers much faster. As a downside, this yields only one ruler for each combination, while the total number of existing rulers remains unknown.

To find all rulers for every known combination of L and M, the software also supports the *known M mode*. In this mode, L is incremented every time the template is finished, and M is set to a previously known value. The template may also be split if the runtime of the previous L was too long.

We used both modes one after the other. First, we searched for the correct L/M with the *stop on result* mode and reached $L = 244/M = 28$. Then, we used the *known M* mode to find all rulers up to $L = 213/M = 25$. We could not reach $L = 244$, as this computation is much harder because all templates need to be checked, and the computation cannot skip any templates.

VI. RESULTS

We performed three different types of searches. Each produced an own set of data. The results are shown in Table 1. Additionally, Table 2 shows some examples for *optimal* rulers. For these calculations, a cluster of ~60 computers was used, providing a few hundred threads. The cluster invested a total single-core CPU time of approximately 443 years for all three searches together. The cluster was largely heterogeneous, consisting of large compute servers and smaller desktop computers. Most machines were equipped with an Intel Core i7-6700K CPU running at 4 GHz. Our implementation with dynamic break level and all results are available online [5].

1. https://bottlepy.org/

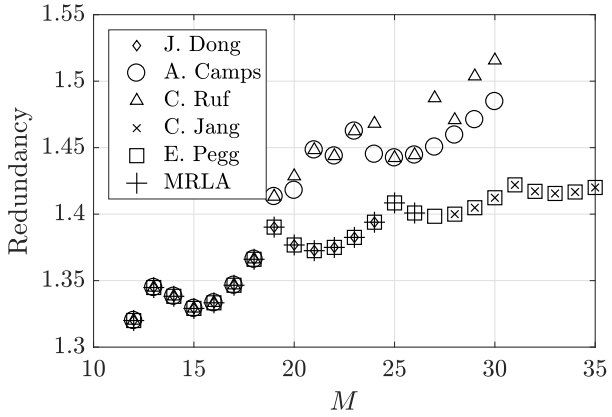

FIGURE 5. Redundancy in dependence of M for *optimal* rulers compared to LRLA results from [7]–[11]. The wrong value for $M = 22$ from [7] has been corrected.

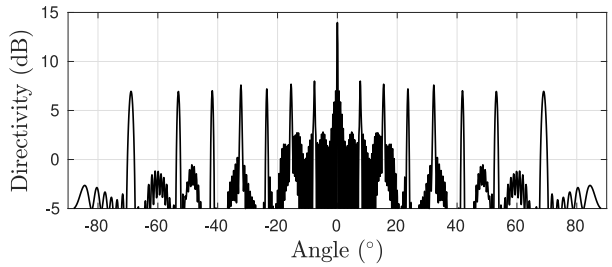

FIGURE 6. Simulated directivity of an arbitrary MRLA with $L = 232$ and $M = 26$. The antenna elements are isotropic and spaced in multiples of $\lambda/2$.

First, we used the Blanton&McClellan algorithm in our parallel implementation and calculated the results up to $L = 190$, in order to verify the results shown in [3] and [4], which were found by the Robison algorithm. As the rulers themselves were not published in both cases, we can only verify that all combinations of L, M, R, and S are correct. R is the redundancy as defined below, and S is the number of rulers found. Next, we used the Robison algorithm to find the correct M for all ruler lengths up to $L = 244$ with the *stop on result mode* (see above). Finally, we searched all rulers up to $L = 213$ with the *known M mode*. The results are also included in Table 1.

The redundancy R in Table 2 and Fig. 5 is defined in [7]:

$$R = \frac{M(M-1)}{2L}. \tag{2}$$

The ruler configuration examples given in Table 2 are represented by differences in the positions of the marks. An exponent defines how many times this distance occurs in sequence. E.g., $\{1^2, 4^2, 3\}$ would expand to the ruler xxxoooxoooxoox, where x represents a mark and o represents the absence of a mark.

Previous results from [2] suggested that if a *perfect* ruler of length L was found with M marks, every *perfect* ruler

TABLE 1. Overview of perfect ruler results.

L	M	S	L	M	S	L	M	S	L	M	S	L	M	S	L	M	S	L	M	S
0	1	1	35	10	5	70	15	2953	105	18	180	140	21	1209	175	23	11	210	26	≥ 8
1	2	1	36	10	1	71	15	1279	106	18	69	141	21	724	176	23	4	211	26	≥ 102
2	3	1	37	11	1339	72	15	425	107	18	35	142	21	327	177	23	1	212	25	2
3	3	1	38	11	487	73	15	194	108	18	14	143	21	189	178	23	1	213	25	2
4	4	0	39	11	181	74	15	60	109	18	4	144	21	96	179	24	21387	214	26	≥ 1
5	4	2	40	11	50	75	15	19	110	18	1	145	21	55	180	24	12259	215	26	≥ 1
6	4	1	41	11	18	76	15	2	111	18	1	146	21	14	181	24	7598	216	26	≥ 1
7	5	6	42	11	2	77	15	3	112	18	1	147	21	5	182	23	1	217	26	≥ 1
8	5	4	43	11	1	78	15	2	113	19	3289	148	21	3	183	23	1	218	26	≥ 1
9	5	2	44	12	2446	79	15	1	114	19	1492	149	22	25950	184	24	1084	219	26	≥ 1
10	6	19	45	12	1057	80	16	2486	115	19	729	150	22	14058	185	24	448	220	26	≥ 1
11	6	15	46	12	342	81	16	1117	116	19	293	151	22	8835	186	24	231	221	26	≥ 1
12	6	7	47	12	119	82	16	399	117	19	187	152	21	1	187	24	115	222	26	≥ 1
13	6	3	48	12	34	83	16	166	118	19	96	153	21	1	188	24	64	223	26	≥ 1
14	7	65	49	12	11	84	16	53	119	19	49	154	22	1341	189	24	40	224	26	≥ 1
15	7	40	50	12	2	85	16	24	120	19	19	155	22	663	190	24	34	225	26	≥ 1
16	7	16	51	13	8159	86	16	2	121	19	7	156	22	397	191	24	13	226	27	≥ 1
17	7	6	52	13	3175	87	16	3	122	19	2	157	22	197	192	24	2	227	26	≥ 1
18	8	250	53	13	1143	88	16	1	123	19	2	158	22	80	193	24	2	228	26	≥ 1
19	8	163	54	13	418	89	16	1	124	20	8679	159	22	36	194	24	1	229	27	≥ 1
20	8	75	55	13	165	90	16	1	125	20	4236	160	22	11	195	25	20082	230	27	≥ 1
21	8	33	56	13	54	91	17	1696	126	20	2090	161	22	4	196	25	11032	231	26	≥ 1
22	8	9	57	13	12	92	17	631	127	20	1074	162	22	3	197	24	1	232	26	≥ 1
23	8	2	58	13	6	93	17	313	128	20	490	163	22	1	198	24	1	233	27	≥ 1
24	9	472	59	14	15990	94	17	106	129	20	255	164	23	18271	199	25	1890	234	27	≥ 1
25	9	230	60	14	6126	95	17	38	130	20	130	165	23	11101	200	25	708	235	27	≥ 1
26	9	83	61	14	2480	96	17	20	131	20	90	166	23	6588	201	25	388	236	27	≥ 1
27	9	28	62	14	903	97	17	8	132	20	46	167	22	1	202	25	207	237	27	≥ 1
28	9	6	63	14	334	98	17	1	133	20	16	168	22	1	203	25	76	238	27	≥ 1
29	9	3	64	14	119	99	17	1	134	20	4	169	23	1047	204	25	57	239	27	≥ 1
30	10	1018	65	14	43	100	17	1	135	21	32617	170	23	494	205	25	31	240	27	≥ 1
31	10	445	66	14	3	101	17	1	136	21	16735	171	23	258	206	25	29	241	27	≥ 1
32	10	152	67	14	6	102	18	1713	137	20	1	172	23	123	207	25	27	242	27	≥ 1
33	10	60	68	14	2	103	18	753	138	20	1	173	23	72	208	25	11	243	27	≥ 1
34	10	10	69	15	7779	104	18	341	139	21	2451	174	23	23	209	26	≥ 31	244	28	≥ 1

TABLE 2. Example rulers for every M up to 28 and the highest L found (optimal rulers, except for $M = 27, 28$).

L	M	R	Array configuration example
1	2	1.00	$\{1\}$
3	3	1.00	$\{1, 2\}$
6	4	1.00	$\{1, 3, 2\}$
9	5	1.11	$\{1^2, 4, 3\}$
13	6	1.15	$\{1^2, 4^2, 3\}$
17	7	1.24	$\{1^3, 5^2, 4\}$
23	8	1.22	$\{1, 3, 6^2, 2, 3, 2\}$
29	9	1.24	$\{1, 2, 3, 7^2, 4^2, 1\}$
36	10	1.25	$\{1, 2, 3, 7^3, 4^2, 1\}$
43	11	1.28	$\{1, 2, 3, 7^4, 4^2, 1\}$
50	12	1.32	$\{1^3, 20, 5, 4^4, 3^2\}$
58	13	1.34	$\{1, 2, 3, 11, 3, 7, 8, 10, 4^3, 1\}$
68	14	1.34	$\{1^2, 3, 5^2, 11^3, 6^3, 1^2\}$
79	15	1.33	$\{1^2, 3, 5^2, 11^4, 6^3, 1^2\}$
90	16	1.33	$\{1^2, 3, 5^2, 11^5, 6^3, 1^2\}$
101	17	1.35	$\{1^2, 3, 5^2, 11^6, 6^3, 1^2\}$
112	18	1.37	$\{1^2, 3, 5^2, 11^7, 6^3, 1^2\}$
123	19	1.39	$\{1^2, 3, 5^2, 11^8, 6^3, 1^2\}$
138	20	1.38	$\{1^3, 4, 7^3, 15^5, 8^4, 1^3\}$
153	21	1.37	$\{1^3, 4, 7^3, 15^6, 8^4, 1^3\}$
168	22	1.38	$\{1^3, 4, 7^3, 15^7, 8^4, 1^3\}$
183	23	1.38	$\{1^3, 4, 7^3, 15^8, 8^4, 1^3\}$
198	24	1.39	$\{1^3, 4, 7^3, 15^9, 8^4, 1^3\}$
213	25	1.41	$\{1^4, 5, 9^4, 19^6, 10^5, 1^4\}$
232	26	1.40	$\{1^4, 5, 9^4, 19^7, 10^5, 1^4\}$
*243	27	1.44	$\{1^3, 4^1, 7^3, 15^{12}, 8^4, 1^3\}$
*244	28	1.55	$\{1^4, 3^1, 7^2, 15^{13}, 8^3, 1^4\}$

with larger L needs at least the same number of marks. Robison [4] already found what he calls gaps. There are certain values for L, which do not produce a *perfect* ruler with M marks, although the combination of $L - 1$ and M does. For example, $L = 137$ requires $M = 20$, while $L = 136$ does not give any results with $M = 20$. What was not shown in [4], is the fact that $L = 136$ can have a *perfect* ruler when increasing M to 21. So there are actually no gaps; it just requires a higher M.

There are a lot of algorithms describing approaches to calculate LRLAs like in [7]–[11]. Our results show, that many of the found LRLAs are actually MRLAs. For example, the two LRLAs for $L = 123$ and $M = 19$ found in [7] are not just MRLAs, they are the only ones for this combination. It also shows that the search presented in [10] and [8] are far superior compared to the others. Fig. 5 gives an overview of the currently calculated LRLAs and our MRLAs. We calculated up to $L = 244$, which results in $M_{min} = 23$. To prove that the results between $M = 23$ and $M = 26$ are *optimal* rulers, we checked that there are no results for higher L values. L has to be calculated as high, such that the corresponding M_{min} is higher than the current M.

As mentioned before, the required CPU time rises exponentially with L, but not monotonically, as shown in Fig. 1. It also shows that the Robison algorithm is about ten times faster than the Blanton&McClellan algorithm. The calculations for Fig. 1 were carried out on a single CPU to avoid runtime differences due to different hardware.

Fig. 6 shows a simulated radiation pattern calculated from the MRLA with $L = 232$ and $M = 26$. The simulation uses isotropic antennas spaced in multiples of $\lambda/2$, which is typical

for such arrays. The size of the array is thus depending on the center frequency of the system according to $\lambda = c/f$. At 2.4 GHz the resulting array would have an aperture (length) of 14.5 m. The achieved half power beam width is 0.32 °.

VII. CONCLUSION

It turned out that an MRLA has a mathematically identical counterpart called *perfect* ruler. As there is no known mathematical rule to design MRLAs or *perfect* rulers, finding them can only be achieved by an exhaustive search, implementing a counterproof showing there is no ruler requiring fewer marks for the same length. This search has a high computational effort. We used two search algorithms and extended them to work with any start template. This allowed us to distribute the problem over dozens of computers and set the start templates to limit the execution time of the individual computations. The gained results and our software is published online and may be used by anyone. The results confirmed previous work and extended it to larger rulers. Additionally, the results show that current LRLA search algorithms can calculate arrays that are at least close to MRLAs or even provide the same results.

REFERENCES

[1] A. Moffet, "Minimum-redundancy linear arrays," *IEEE Trans. Antennas Propag.*, vol. 16, no. 2, pp. 172–175, Mar. 1968.

[2] K. A. Blanton and J. H. McClellan, "New search algorithm for minimum redundancy linear arrays," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 1991, pp. 1361–1364.

[3] P. Luschny. (2018). *Perfect and Optimal Rulers*. [Online]. Available: http://www.luschny.de/math/rulers/prulers.html

[4] A. D. Robison. (2014). *Parallel Computation of Sparse Rulers*. [Online]. Available: https://software.intel.com/en-us/articles/parallel-computation-of-sparse-rulers

[5] F. Schwartau, Y. Schröder, L. Wolf, and J. Schoebel. (2020). *MRLA Search Results and Source Code*. [Online]. Available: http://dx.doi.org/10.21227/cd4b-nb07

[6] P. Luschny. (2018). *Perfect and Optimal Rulers*. [Online]. Available: http://www.luschny.de/math/rulers/introe.html

[7] J. Dong, Q. Li, R. Jin, Y. Zhu, Q. Huang, and L. Gui, "A method for seeking low-redundancy large linear arrays in aperture synthesis microwave radiometers," *IEEE Trans. Antennas Propagat.*, vol. 58, no. 6, pp. 1913–1921, Jun. 2010.

[8] C.-H. Jang, F. Hu, F. He, J. Li, and D. Zhu, "Low-redundancy large linear arrays synthesis for aperture synthesis radiometers using particle swarm optimization," *IEEE Trans. Antennas Propagat.*, vol. 64, no. 6, pp. 2179–2188, Jun. 2016.

[9] A. Camps, A. Cardama, and D. Infantes, "Synthesis of large low-redundancy linear arrays," *IEEE Trans. Antennas Propagat.*, vol. 49, no. 12, pp. 1881–1883, Dec. 2001.

[10] E. Pegg, Jr. (2020). *Sparse Rulers—Wolfram Demonstrations Project*. [Online]. Available: https://blog.wolfram.com/2020/02/12/hitting-all-the-marks-exploring-new-bounds-for-sparse-rulers-and-a-wolfram-language-proof/

[11] C. S. Ruf, "Numerical annealing of low-redundancy linear arrays," *IEEE Trans. Antennas Propag.*, vol. 41, no. 1, pp. 85–90, Jan. 1993.