# Parallel Construction of Orthogonal Arrays

Vincent Brouérius van Nidek
3207900

Supervisors:
Rob Bisseling
Eric Schoen
Pieter Eendebak

April 5, 2009

# Contents

# Chapter 1

# Introduction

This report is on the generation of orthogonal arrays by a parallel computer program. The research has been done for the master Scientific Computing at the Department of Mathematics at Utrecht University. The assignment came from the "Nederlandse organisatie voor toegepast-natuurwetenschappelijk onderzoek (TNO)" (Dutch organization for applied scientific research). TNO is a Dutch organization which aims "To apply scientific knowledge with the aim of strengthening the innovative power of industry and government", see the website `http://www.tno.nl`. TNO has bundled its research topics into five core areas:

- Quality of Life
- Defense, Security and Safety
- Science and Industry
- Built Environment and Geosciences
- Information and Communication Technology

The research was conducted at the statistics section, part of the business unit "Industrie en techniek" (Science and industry).

## 1.1 Orthogonal Arrays

In many areas of research, tests have to be done to find a relation between some input parameters and the output of an experiment. A straightforward method would be to test every possible combination of parameter settings. This can become very time and capital intensive when the number of parameter settings increases. Orthogonal arrays can be used to investigate the influence of parameters with less experiments.
An orthogonal array represents a statistical method of experimenting, given by a matrix of numbers. It gives a systematic way to sample the test domain, without checking every possible combination. This report is on the generation of orthogonal arrays from the set of numbers describing them. At this moment, several methods exist to do this such as by integer linear programming (Bulutoglu and Margot, 2008). Another method only applies to a special group of arrays, namely when one of the variables has to be varied in a prime number of steps (Addelman and Kempthorne, 1961). Other limitations to a design can be, that the set of

experiments contains only unique entries (Bush, 1952). A drawback of some methods is, that they give one or some possible solutions, while the method described in this report gives *all* the solutions.
Orthogonal arrays are mainly used in the design of experiments; obtaining parameter influence with a limited set of experiments or obtaining as much information as possible with a given number of experiments. Orthogonal arrays can be used to describe a design of experiments, but they can also be used for noise reduction (Delsarte, 1973).
The name orthogonal array can be confusing, since "orthogonal" usually means that the inner product of two vectors is zero. This does not apply to orthogonal arrays and is not possible. The orthogonality is because every column contains some unique information of the settings; the name is first mentioned in Rao (1947b).

## 1.2 Program

Orthogonal arrays can be generated from the numbers describing an array. In order to achieve this, a program has been written, which produces orthogonal arrays which satisfy the initial conditions. At the beginning of this research, a program written by Ruben Snepvangers has been used (Snepvangers, 2006). This program has produced several orthogonal arrays, but it has its limitations due to the depth-first approach. When a problem occurs during the execution of the program, all intermediate work is lost. Due to a very limited output it was sometimes difficult to distinguish between a crashed program and a (very) long run time. Therefore a new program was needed, which gave a better insight into the progress. A different approach in the generation of orthogonal arrays based on breadth-first search enables the saving of intermediate results. Results from previous runs can now be used to restart the program and continue. The theory of orthogonal arrays is explained in chapter 2 and 3. Chapter 4 gives the details on the implementation of the algorithm for the generation of orthogonal arrays. This chapter also gives the background about the differences between the old and new program. In chapter 5 the results obtained with the program are given and are they discussed in chapter 6. Based on these results and conclusions, recommendations on research and program improvements are done in chapter 7.

## 1.3 Objective

Based in experiences from using the previous program, we come to the conclusion that a new program is needed. The objective of this research is to produce a program, which gives the complete set of solutions for any input. Since the calculation time can increase sharply for large cases, the program has to be able to run i parallel. Therefore, the program must take advantage of multiple processors and generate speed-up on a distributed system. In contrast to the program written by Snepvangers, the breadth-first approach will be used.
Furthermore, to demonstrate the use of the new program, some open cases will be calculated. These are arrays for which (all) the solutions are currently unknown.

# Chapter 2

# Theory

An orthogonal array gives a schematic representation of an experiment. This chapter describes the theoretical background. First, an introduction to orthogonal arrays and their characteristics, then the classification of the different arrays is given in section 2.1. The strength of an orthogonal array is explained more extensively in section 2.1.1. Although it is possible to generate a huge amount of solutions of an orthogonal array for a given case, only several classes exist. Section 2.2 gives the details on such classes and how a unique array can be selected from each class.

## 2.1 Orthogonal array parameters

An orthogonal array describes a set of experiments and shows which parameter settings need to be used. For the ease of notation and general use of the arrays, each parameter setting is given a number, a zero or positive integer. The value represents a setting of the variable, which could be "hot", $T = 100^o\,C$, $c = 100\,g/L$, $m = 10.1\,g$ *etc.* Table 2.1 gives an example of an orthogonal array. The orthogonal array is a matrix with 8 rows and 4 columns. Every row in an orthogonal array represents an experiment, so this array describes a set of 8 experiments; the number of **runs**. The symbol used for the number of runs is $N$. Every column stands for a parameter which can be varied and is called a **factor**. Every number in the array gives the parameter setting for the corresponding experiment and variable and is called an **element**. In this design, only 0 and 1 elements appear, meaning that every column has two **levels**. Since this array only contains factors with two levels, it is called a **pure array**. Arrays with different number of levels per factor are called **mixed arrays**.

**Table 2.1:** Example of a pure orthogonal array

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |

Orthogonal arrays can be described by a small set of parameters, where the number of runs is the first. The second part of the parameters consists of a set, describing the number of factors and the number of levels per factor. The factors for the columns are described by $S = s_1^{q_1} \cdot \ldots \cdot s_n^{q_n}$, where $s_1 > s_2 > \ldots > s_n$ and $q_i \geq 1$. The number of factors is equal to $\sum_{i=1}^{n} q_i$. The final describing number is given by the **strength**, with the symbol $t$. For an orthogonal array of $N$ runs with strength $t$, any $N \times t$ subarray contains every possible $t$-tuple based on $S$ equally often. In the example of table 2.1 every 2-tuple, $\{0,0\}$, $\{0,1\}$, $\{1,0\}$ and $\{1,1\}$, appears 2 times for every pair of columns. The arrays in table 2.1 and 2.2 have strength 2, while the array in table 2.3 has strength 3. In this report the notation for orthogonal arrays is given by $OA\,(N,S,t)$. The example in table 2.1 can than be represented by $OA\,(8,2^4,2)$.

In the description $S$, a set of columns with the same number of levels is called a **section** and is given by an element from $S$, $s_i^{q_i}$. By definition, the columns of a section are contiguous. The first $t$ columns of an orthogonal array are called the **root** and it is used as a starting point for the array generation more on this in chapter 3. With the limitation on $S$ given earlier and the limitation that only arrays in lexicographic minimum form is used, only one root can be constructed.

**Table 2.2:** Example of an orthogonal array

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 0 | 2 | 2 | 2 |
| 1 | 0 | 1 | 2 |
| 1 | 1 | 2 | 0 |
| 1 | 2 | 0 | 1 |
| 2 | 0 | 2 | 1 |
| 2 | 1 | 0 | 2 |
| 2 | 2 | 1 | 0 |

Another important number is the **index**, $\lambda$, of an array, which tells how often a line is repeated in the root. The root is the first $t$ columns in an array and is unique for a given design, see section 3.1 for more details. In table 2.2, the root consists of the first two columns, and it can be seen that every line occurs only once; hence the index is one, or *unity*. The index can be calculated by: $\lambda = \frac{N}{\prod_{i=1}^{t} s_i}$. An array with an index of two is created by doubling the number of runs from an index unity; $OA\,(18,3^4,2)$ has an index of two. By doubling the number of runs, a new root is created, which needs to be extended further. More on the lexicographic form of an array can be found in section 2.2.

### 2.1.1 Strength of an array

The strength of an array tells which correlation between variables can be measured; a strength of two can measure the main effect between two variables. With strength of three, the interaction of factors does not disturb the main effects. Higher strengths can measure more interactions of parameters with less influence from other parameters. Generally, more experiments are needed to achieve this. A strength of $t$ means, that any $t$-tuple of elements should

occur equally often in the design. The frequency of a tuple is denoted by $\nu$. For homogeneous arrays, the frequency of a tuple is determined by the number of runs in an experiment, $N$, the number of factors per element, $s_i$, and the strength, $t$. The formula for the frequency is:

$$\nu = \frac{N}{s^t} \tag{2.1}$$

For mixed arrays, multiple $s$ values exist and an alternative formula is given by Hedayat et al. (1999). The denominator in equation (2.1) is now given by the product of the combination of $t$ columns from $S$. Equation (2.2) gives the alternative for the denominator of equation (2.1) when a mixed array is considered:

$$\nu = \frac{N}{s_1 \cdot s_2 \cdot \ldots \cdot s_t} \tag{2.2}$$

Equation (2.2) shows that a pure array is only a special case of a mixed array and with all number of factors equal $\prod_{i=1}^{t} s_i = s^t$. The solution of $OA\left(8, 2^4, 3\right)$ is given in table 2.3. In any combination of three columns, a 3-tuple is encountered only once, as required by equation (2.1).

**Table 2.3:** Example of an array with strength 3

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

## 2.2 Isomorphism classes of arrays

This report is on the generation of orthogonal arrays from their description by column-wise extension. This extension method can result in multiple solutions, which can be grouped in isomorphism classes. Arrays within a class can be transformed into each other by means of permutations, whereas for arrays of different classes this is not possible. Arrays within the same class are said to be **statistically equivalent** or belong to the same **isomorphism class**. A property of arrays within one class is, that they describe the same set of experiments. For this research only one array per class is needed, the selection criteria are described in section 2.2.2.

### 2.2.1 Permutations

When applying permutations on an orthogonal array, it remains in the same class. Three types of permutations are allowed; column, row and level permutations. The description of each type of permutation is given by:

**Column permutations** Within a section, columns can be freely permuted, whereas permutations between different sections are not allowed. A column represents the setting of a given parameter for the different experiments. Within a section, every factor has the same number of levels. Since the order of the factors is arbitrary, swapping columns does nog change the actual set of experiments described.

**Row permutations** For row permutations, every row from the design can be swapped with another in the array. Since a row represents an experiment, swapping rows will only change the order in which the experiments would be performed. Therefore a row permutation yields a different orthogonal array, but does not change the class it belongs to.

**Level permutations** For each column, the settings for one parameter are given. The assignment of the numbers for each setting can be chosen freely. Therefore the numbers for each factor can be freely permuted, without affecting the nature of the experiment. The total number of level permutations that can be performed, is the combination of all level permutations of all columns.

### 2.2.2 Lexicographic ordering

For this research, from every class, the array in **lexicographic minimum in columns**(lmc) form is used as representative. Lexicographic ordering is also called natural ordering or dictionary ordering. In table 2.4 two fractions of arrays are given, where table 2.4(a) is not lexicographically ordered. When the elements from a row are seen as a number, in this case binary, the ordered version can be obtained by sorting these numbers in ascending order. When two rows are exactly equal, their order does not matter. This allows an *unstable* sorting algorithm to be used. The result of the sorting action gives the array as seen in table 2.4(b).

**Table 2.4:** Comparison of fragments of orthogonal arrays

(a) Lexicographic unsorted fragment

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |

(b) Lexicographic sorted fragment

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |

For the array generation algorithm used in this research, the array that is in lmc form is used. In order to find this array, the permutations mentioned in section 2.2.1 are executed. Every combination of the permutations is used on the array and every result is compared to the original array. When every result is lexicographically more than the original, the array is in the lexicographic minimum form. The details of this test is explained in section 3.3 and the implementation in 4.3.

# Chapter 3

# Algorithm

Several methods are known to generate orthogonal arrays, but they usually only yield one solution, while multiple solutions could exist. Usually they construct arrays following special rules. Here, a different approach is taken; all possible solutions are tested and all arrays that pass form the the group of solutions. As a starting point for the algorithm the root is used. From the parameters of an array, the array root can always be generated. Then, the root is extended column wise. Each extension has to undergo a lexicographic minimum test, the ones that pass are unique in their class and form the basis for further extension. Section 3.1 describes how the root can be calculated and section 3.2 gives the details on how an extra column can be added to an existing array. In section 3.3 the details on the lexicographic minimum test are given.

## 3.1 Root generation

The algorithm used for this research extends an orthogonal array with an extra column. The starting array for this algorithm is the root and it contains $t$ columns. The first column is monotonically increasing, with each element repeated $\frac{N}{s_1}$ times. For the second column smaller cycles are used with each unit repeated $\frac{N}{\prod_1^2 s_i}$ times and the partial column that is created is repeated until the column is filled. If the strength is larger than two, the third column is constructed similar to the second, only now each element is used $\frac{N}{\prod_1^3 s_i}$ times, *etc.* Table 3.1 gives an example of a root; in the first column each element is repeated $\frac{8}{4} = 2$ times, in the second column it is $\frac{8}{4 \cdot 2} = 1$ times.

**Table 3.1:** Root for the design $OA\,(8, 4 \cdot 2^n, 2)$.

| | |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |
| 2 | 0 |
| 2 | 1 |
| 3 | 0 |
| 3 | 1 |

## 3.2 Array generation

After the root has been constructed, the array is generated by column wise addition. For every column that has to be added, all possible extensions are generated and tested. All extensions that pass the test form the group of solutions. These solutions form the new starting point where a new column will be added to. This breadth-first approach finds every valid orthogonal array with one column more than before.

**Algorithm 1** Design generation algorithm
**input**: root $R$ and characteristic numbers, **output**: set of solutions

```
Q = R
for all columns to be extended do
  X = Q, Q = 0
  for all arrays in Q do
    Column Extension (input X_p, output C(X_p))
    for all arrays in C(X_p) do
      LMC TEST (input C(X_p)_q, output ξ (ξ = 0; failed, ξ = 1; passed)
      if ξ = 1 then
        Q = Q + C(X_p)_q
      end if
    end for
  end for
  d = |Q|
  if d = 0 then
    done
  end if
end for
```

The algorithm takes a given array and adds a column to it, element by element. Algorithm 1 shows the method used (Schoen and Nguyn, 2007). First the root, $R$, is copied to the current set of arrays which needs to be extended, $Q+$. The algorithm input contains the desired number of columns, as long as this is not reached, more columns are added. The "working" set of solutions is $X$ and $Q$ is cleared, so the new solutions with one column more can be put into it. $X_p$ is one of the arrays from the set $X$, which results in an extended set $C(X_p)$ after the array is extended with one column. From the extended set, each element $C(X_p)_q$ has to undergo the lmc test. The arrays that pass the test are copied to the set $Q$. If no more solutions exist, $d$ equals zero and the algorithm is terminated.

The algorithm terminates when the desired number of columns is reached or when no more solutions can be found. For pure arrays, the Rao bound (Rao, 1947a) predicts if the desired number of columns can be constructed. The Rao bound calculates a value from the numbers describing orthogonal arrays. Valid orthogonal arrays have a Rao bound rest of zero and higher, negative values indicate a design which can not be constructed.

## 3.3 Lexicographic minimum test

In order to get one array per class, the array in lexicographic minimum in columns form is chosen. The method to select this is shown in algorithm 2; a given array undergoes a combination of level, row and column permutations and the result is compared to the original input. If the permutations result in an array which is lexicographic less, the input array is not the minimum. When an array is compared to a permuted version and the result is lexicographic less, no more permutations have to be done. If an array has undergone every possible combination of permutations and no array in a less form is found, one is sure it is an array in the lexicographic minimum form.

**Algorithm 2** Lexicographic Minimum Test
**input**: array, **output** $\xi$ ($\xi = 0$; failed, $\xi = 1$; passed

```
cpy = array
for all row permutations do
  Perform row permutation
  for all column permutations do
    Perform column permutation
    for all level permutations do
      Perform level permutation
      if cpy <array then
        ξ = 1
      else
        ξ = 0
      end if
    end for
  end for
end for
```

# Chapter 4

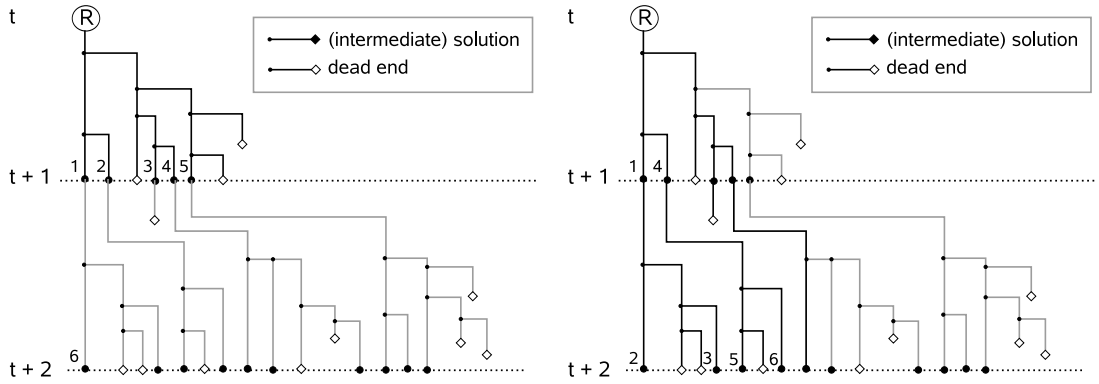
# Parallel Implementation

In chapter 3 the algorithm which is used in this research is described. The program by Snepvangers implements this literally, while in this research a variation is used. The program used for this research is written in C, while it contains elements of C++ where they are more suitable. Examples of this are an implementation of a linked list or the flexibility and performance of sorting data. The details on the exact implementation are given in this chapter. First, the breadth-first approach is described in section 4.1. Section 4.2 gives the details on how the extension with an extra column is implemented. The lexicographic minimum test implementation is given in section 4.3. In the implementation special attention has been paid to the sorting of rows in the lmc test. Therefore a section is dedicated to this part; section 4.4. Finally, section 4.5 describes how the communication between the different nodes is done, when the arrays are generated in parallel.

## 4.1 Breadth-first approach

The breadth or depth first approach is best described visually, when all the solutions are put into a tree. The root of the tree is given by the root of the design and the leaves are the solutions with the maximum number of columns. Intermediate terminated branches indicate incomplete extensions. The depth first approach is the method which continues into the leaves as soon as possible, while the breadth-first approach finishes each level of branches completely.
In figure 4.1 an overview is given of the difference between the breadth and depth first approach. Figure 4.1(a) shows an array which is extended with two columns; the first extension results in five solutions and the second extension in ten. The order of the first six solutions in which the arrays are found, is indicated by the number next to the filled dot. The color of the lines also indicates the progress of the algorithm.

What can be clearly seen from figure 4.1 is that eventually both approaches give the same set of solutions. For the depth first approach it would be possible to track the progress in some way, but this is (much) more difficult than for the breadth-first approach. At this moment the progress is tracked in two ways; the first is displaying which column is being filled. The second way shows which solution from the previous extension is currently used. In the output a time measurement is included as well, which makes it possible to estimate the remaining time needed.
In the breadth-first program, all (intermediate) solutions are stored as soon as all possible




(a) Breadth first (b) Depth first

**Figure 4.1:** Difference between depth and breadth-first

extensions at that level have been checked. When the solutions are stored, one can be sure that all solutions of one column addition have been found. The intermediate results are directly available and can already be used. They can also be used as a starting point for further extension because of the restart option in the program.
The breadth-first approach can be seen in the code in the main file `mpi_frame_alt.cpp` in the `main` function. For the master processor, first a loop for all columns is given. Inside that loop a second loop is present, to extend all solutions from the previous extension with one more column. After all solutions have been found, they are saved and it proceeds with the next column.

## 4.2 Column Extensions

The implementation for the generation of extra columns is easily described, but a general method is complicated. The extension with a column is done element-by-element, starting with the first row. The generation of a column can be seen as a branched network or a tree; the starting point is the first element and the end points are valid extensions. The first element of every column is always zero, but after this, multiple possibilities exist for every element. Every element with more options generates one or more branches. Every branch that is created, can generate newer branches, when another element with multiple options is encountered. A branch terminates when a column is completely filled or when no more elements can be added due to the constraints.
The algorithm used to generate the column extensions is given in algorithm 3. Since for every element the lowest value possible is used for the first attempt and higher values are used later in an ascending order, the extensions are produced in a lexicographically ordered way. The code for this algorithm can be found in the function `extend_array` in the file `extend.cpp`.

## 4.3 Lexicographic minimum test

In section 3.3 the general algorithm behind the lexicographic minimum test is given. From the three types of permutations that are mentioned, the row permutations are not performed exactly as mentioned, but a faster alternative is used.. For the speed of the implementation,

**Algorithm 3** Column generation algorithm
**Input** $X_i$, **output** $C(X_i)$

```
first element zero
continue with next element
while not done do
  if current element is empty then
    fill in lowest value possible
    if multiple options possible then
      store current location
    else if no options then
      get last stored location
      if stack is empty then
        done
      end if
    end if
  else
    get next option for element
    remove last element
    if multiple options possible then
      store current location
    else if no options then
      get last stored location
      if stack is empty then
        done
      end if
    end if
  end if
end while
```

it is important to fail the test as soon as possible. Therefore a variation is used instead of row and ordinary level permutations. Each row in the permuted array is selected one-by-one, for each row, its value in each column will be permuted to zero. For all other values in the columns, the normal level permutations will be performed. After each operation, the array is sorted and the sorted array is compared to the original input array. This method has the same output as the original algorithm but will sooner result in the rejection in the lmc test. Since the first row of an orthogonal array always consists of zeroes only, this can be seen as putting a row at the first row and level permute the rest. This method generates arrays that are more in lmc form than would be obtained by the (random) original permutations. When an array can be rejected sooner, this saves calculation time.
In the file `permutations_alt.cpp`, the first function in the lmc test is `lmc_test` and `select_permutations`, which calculates the next level permutations. The permutation itself is performed by `perm_levels`, which is a recursive function. The function calls itself for each next column to give the correct level permutations. The selected row is put into the first position by `insert_map_zero`. After a level permutation is performed, the recursive function `perm_multi_col` is called and generates the column permutations. The function calls itself for every new section of columns it encounters. The column permutations are performed by `perform_col_perm` and the result is finally sorted in `sort_rows`.
Every array that passes the test, has to undergo each permutation. The number of column permutations is given by equation (4.1); equation (4.2) gives the number of level permutations (Schoen and Nguyn, 2007).

$$|g| = \prod_{i=1}^{n} q_i! \tag{4.1}$$

$$|h| = N \prod_{i=1}^{n} ((s_i - 1)!)^{q_i} \tag{4.2}$$

The maximum number of permutations that an array has to undergo is given by the product: $|g \cdot h|$. This is only for an array in lexicographic minimum form, for non-minimum arrays, this is the upper limit. It can be easily seen that even for small arrays the number of permutations can be large. Taking the array $OA\left(12, 3 \cdot 2^3, 2\right)$ the number of permutations is: $|g \cdot h| = 12 \cdot \prod_{i=1}^{2} q_i! \cdot \prod_{i=1}^{2} ((s_i - 1)!)^{q_i} = 144$. A slightly more complicated array of $OA\left(27, 9 \cdot 3^4, 2\right)$ already amounts up to $|g \cdot h| = 27 \cdot 8! \cdot 4! \cdot 2^4 = 73728$.

## 4.4 Row sorting

For the sorting of the arrays, the C++ `sort` is used. This function has a low worst case run time, due to the combination of quicksort and heapsort. It is guaranteed that the complexity for worst case is $\mathrm{O}\left(n \log n\right)$. Quicksort has a worst case complexity of $\mathrm{O}\left(n^2\right)$ and in most situations in practice, the `sort`-routine has far better performance. The base function is `sort_rows` but the improved sorting functions can be found in `lmc.cpp`
The arrays are stored as a one dimensional array in the computer, split in columns. Sorting the rows directly would require a (relatively) high number of switching individual cells. Therefore the rows are not sorted directly. First a **rank** is calculated, which indicates how a row will

be sorted. The values in a row $r$ up to column $c$ are given by $R_{r,c}$ and the values from $S$ up to column $c$ by $S_c$, both in row vectors. The rank $\Re_{r,c}$ is than calculated by:

$$\Re_{r,c} = \sum_{l=1}^{c} \left( R_{r,c} \cdot \prod_{m=l+1}^{c} S_m \right) \tag{4.3}$$

First the ranks will be sorted ascending and afterwards, the rows are put in the same order. This only needs the sorting of two columns, the rank and the original row number, instead of a complete row.
Due to the special structure of orthogonal arrays, further improvements are possible. For *every* orthogonal array in LMC form, the root looks the same, regardless of permutations or sorting it has undergone. For arrays with an index of 1, it is therefore always enough to calculate the rank of the first $t$ columns, $\Re_{r,t}$, and sort according to it. For arrays of higher indexes, a second sorting step is needed. The semi sorted array is sorted in "blocks" of the root with the same number of rows as the index. Now each block must be sorted with a rank of the full row, except for the first $t$ columns, since they were equal. After a block is sorted, it can be compared to the original array and when a difference in lexicographical order is found, a decision can be made to reject the array or not. As long as no difference is found, the next block can be sorted. This reduces the amount of sorting needed to the minimum necessary to make a decision. In most of the cases a rejection can be found, based on a partially sorted array.

## 4.5 Communication

The communication in the program is handled via the message passing interface (MPI[1]). At TNO a Cray[2] supercomputer with MPI installed was available. Another type of parallel computing was available as well; a condor[3]network. This requires a different approach of task distribution and recommendations on adapting the program for this type of network is given in chapter 7.

The processors in the (super)computer network are divided into two categories; the master and slaves. The master is the central processor which coordinates all the tasks and distributes them between the slaves. The only communication is between the master and a slave, between the slaves no communication takes place. Figure 4.2 shows a schematic representation of the communication. The amount of data sent by the master is (much) smaller than the response from the slaves. The master node generates the column extensions of every array, while the slaves perform the lexicographic minimum test. This was done, since the lexicographic minimum test would take considerably longer than the generation of arrays.
The main functions of the master are given in the `main` function and the `extend_array` function in the file `extend.cpp`. In the file `communication_alt.cpp` the most important functions for the communications between master and slaves is given. The function `slave_code` is the most important function for the slaves; they report their status to the master and wait for a

[1] See official website for more details: `http://www.mpi-forum.org`
[2] A wellknown brand of supercomputers, see `http://www.cray.com` for more details
[3] Official website `http://www.cs.wisc.edu/condor/`

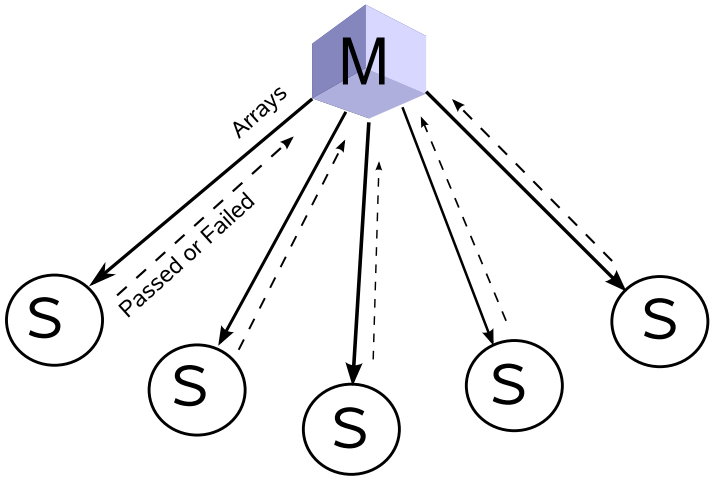



**Figure 4.2:** Communication between master and slaves

new job. The function `lmc_test` tells the master what to do with an array which needs to be tested. In the case of one processor, it will be checked by itself, but when more processors are used, it is sent to a slave. In `send_array_base` the constant part of an array is sent only once and the extended column is the only data needed by the slaves. This reduces the amount of data which needs to be sent.

# Chapter 5

# Results

The first and most important result of this research is, that a successful program has been written. For testing purposes, produced arrays are compared to known solutions from Brouwer (2008). Since all arrays matched with various settings, all arrays are assumed to be correct. The program has been used to calculate some open cases, the results are given in section 5.1. The tests for this computer were performed on the Huygens[1] parallel computer at SARA in Amsterdam, consisting of Power 6 nodes. A characterization of the system is given by means of a benchmark, for which the results are given in section 5.2. Since the program is able to run on multiple nodes, an analysis has been done about the parallel speed-up. The results of these runs are presented in section 5.3. The program has been written to replace the program by Snepvangers, a comparison of runtimes is given in section 5.4. At the current state of the program, it is difficult to identify the best place for further improvements. Section 5.5 gives an overview of how the calculation time is split over various tasks. Section 5.6 gives an overview of the distribution of the times per lmc test. This will help point out areas for future accelerations.

## 5.1 Open cases

The basic target for this research was to generate a faster and more functional program. As an optional objective the calculation of some open cases was chosen. They could not be calculated with the current techniques and could put the new program into good use. Table 5.1 gives the number of extensions found for each open case for the given number of columns. For the array $OA(27, 3^{13}, 2)$ it was known that 68 arrays existed, but no intermediate results were known. The results confirm this number but were not influenced by the prior knowledge of the final number of arrays.

## 5.2 MPI Benchmark

The results obtained in this research depend heavily on the hardware architecture used. Therefore a benchmark program was used, `MPIBench` from the `MPIEdupack`[2] (Bisseling, 2004). This benchmark quantifies the speed of the parallel computer with numbers. They indicate the

[1] For more information, see `https://subtrac.sara.nl/userdoc/wiki/huygens/description`

[2] Obtained from the website: `http://www.math.uu.nl/people/bisseling/software.html`

**Table 5.1:** Number of (intermediate) solutions for the open cases

| Column | $OA\left(16, 4 \cdot 2^{12}, 2\right)$ | $OA\left(16, 2^{15}, 2\right)$ | $OA\left(27, 9 \cdot 3^{9}, 2\right)$ | $OA\left(27, 3^{13}, 2\right)$ |
|---|---|---|---|---|
| 3 | 3 | 3 | 2 | 9 |
| 4 | 10 | 5 | 7 | 711 |
| 5 | 28 | 11 | 10 | 187188 |
| 6 | 65 | 27 | 13 | 922548 |
| 7 | 110 | 55 | 12 | 157829 |
| 8 | 123 | 80 | 9 | 21688 |
| 9 | 110 | 87 | 5 | 9793 |
| 10 | 72 | 78 | 4 | 3766 |
| 11 | 38 | 58 | | 1252 |
| 12 | 15 | 36 | | 341 |
| 13 | 8 | 18 | | 68 |
| 14 | | 10 | | |
| 15 | | 5 | | |

characteristics of the parallel computer used. Table 5.2 shows the results of the benchmarks with two different settings. The benchmark is run twice, once with all the processors on one node and once with all the processors spread over two nodes.

**Table 5.2:** Results of the MPIBenchmark program of the Huygens computer

| Property | 1 Computer | 2 Computer |
|---|---|---|
| $p$ | 16 | 32 |
| $r$ | 4150 | 4131 |
| $g$ | 0.01 | 0.18 |
| $l$ | 52.6 | 1392 |

In table 5.2, four parameters are given, $p$ is the number of processors used to run the program. The parameter $r$ gives the computing rate of the computer, measured in $Mflop/s$. The parameters $g$ and $l$ describe the time needed to communicate data, given in $\mu s$. It is assumed that the relation between transfer time, $T$, and amount of data, $h$, is linear and given by Bisseling (2004):

$$T(h) = hg + l \tag{5.1}$$

The unit of the amount of data is *word*, which is in this case 8 *bytes*. From table 5.2, it can be concluded that a large difference exists when the processors are within a node or not. This is due to the design of the computer; it consists of nodes with 16 processors each. Communications within the same node are much faster than between nodes.

## 5.3 Parallel speed-up

Initially the calculation times for complex arrays were too long and a parallel approach was needed. At this moment the calculation times have been reduced considerably. But for the future cases with even higher complexity or sizes, a good parallel speed-up is still needed. In order to keep the comparison on the calculation part as much as possible, saving of the arrays

is disabled and the output is kept to a minimum by putting the `loglevel` at 1.
The efficiency of the parallel setup, $\eta$, is given in equation (5.2). It is calculated by comparing the single cpu run time, $T_1$, with the run time with multiple cpus, $T_m$. The final performance is given, by accounting for the number of cpus involved, $m$.

$$\eta_m = \frac{T_1}{m \cdot T_m} \tag{5.2}$$

Figure 5.1 shows the efficiency when more processors are used for the first columns of $OA\left(27, 9 \cdot 3^9, 2\right)$. The results show a dramatic decrease in the efficiency of the processors when using multiple processors. For reference the black dotted line shows the line $y = 1/x$. For jobs with efficiencies below this line, the total calculation time is *larger* than the time with one processor. Thus representing a slow-down instead of speed-up. Since all points are below this line, the slow-down of this program is dramatic. The speed-up even becomes worse when the case of $OA\left(16, 4 \cdot 2^{12}, 2\right)$ is considered. The efficiency drops sharply and therefore results in unacceptable long run times.

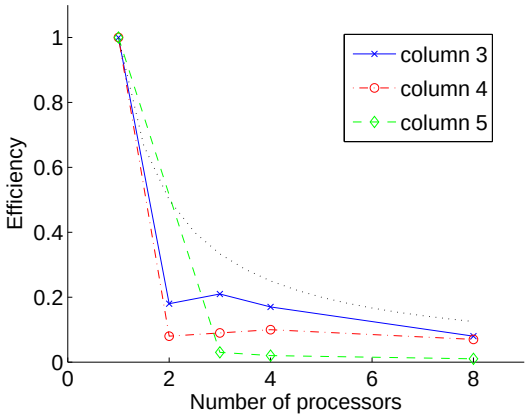



**Figure 5.1:** Parallel speed-up for $AO\left(27, 9 \cdot 3^4, 2\right)$

## 5.4 Calculation time comparison

This research was initiated as a replacement for an existing program, therefore it is compared to its predecessor (Snepvangers, 2006). For both programs only one cpu is used, since they give the highest efficiency. Table 5.3 shows the cumulative times for several runs for both programs. The table shows a very mixed result, both programs perform better than the other in some cases. The times for the arrays of $OA\left(48, 4 \cdot 2^6, 2\right)$ differ one order of magnitude in value. The build-up of the time is remarkable; adding the first column takes a relatively large amount of time, while adding the other columns is considerably faster. For both cases with 16 runs the program by Snepvangers took more than 50 minutes and were terminated. Although more extensions are possible for the arrays of $OA\left(16, 4 \cdot 2^{12}, 2\right)$, no more columns could be added by Snepvanger's program.

**Table 5.3:** Calculation time comparison with program from Snepvangers (2006)

| Design | time `oaextend` $[s]$ | | | | time Snepvangers (2006) $[s]$ |
|---|---|---|---|---|---|
| | 4 | 5 | 6 | 7 | total |
| $OA\left(16, 2^{12}, 2\right)$ | | | | 9 (total) | $> 3000$ |
| $OA\left(16, 4^{\cdot 12}, 2\right)$ | | | | 5 (total) | $> 3000$ |
| $OA\left(48, 4 \cdot 2^6, 3\right)$ | 2426 | 2444 | 2447 | 2455 | 260 |
| $OA\left(48, 6 \cdot 2^5, 3\right)$ | | 176 | 514 | 540 | 255 |

## 5.5 Calculation time breakdown

At this moment an all-round program has been written, which performs reasonably well for most cases. In order to identify the bottlenecks an overview of the distribution of the calculation time is needed. In this breakdown minor tasks such as initialization and input/output are not considered. During normal operation some time has to be added for these operations. Four types of tasks are distinguished: array generation, communication, lmc tests for valid and invalid extensions. All times are measured except the communication time, which is calculated from the numbers by the benchmark. First a breakdown is given for the case on a single cpu, then an overview for multiple cpus is given. For detailed tables and numbers from these experiments, see appendix B.

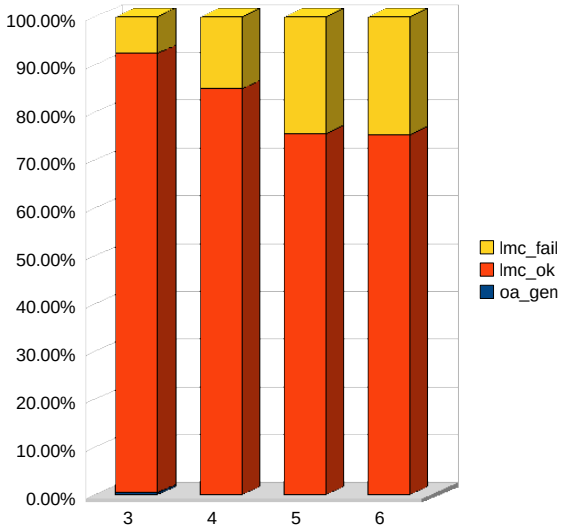
### 5.5.1 Single cpu breakdown

For two open cases a breakdown of the calculation time is done, shown in figure 5.2 and 5.3. In the graph the relative distribution between the time needed for the generation of all possible arrays, `oa_gen`, for all arrays passing the lmc test, `lmc_ok`, and for all the arrays failing, `lmc_fail`. The table show how much time was needed to complete each column addition and how many arrays were generated in total.
From figure 5.2 it can be seen that the time for generating the arrays can be neglected. Only a small amount of arrays is generated and from table 5.1 it can be seen that a small portion passes the lmc test. A division of these numbers shows an increase of the lmc test time per array from $11.0s$ roughly linear up to $48.6s$. Because of the high number of arrays compared to the time needed, the time per array that did not pass the lmc test is relatively small.

Figure 5.3 shows a completely different picture than figure 5.2; now the generation of the arrays and the lmc test for arrays that failed the test are dominant. This is due to the high number of arrays generated, the actual generation time per array is similar to the previous case. The time for an array to pass the lmc test is considerably lower though, a difference of five orders of magnitude is observed.

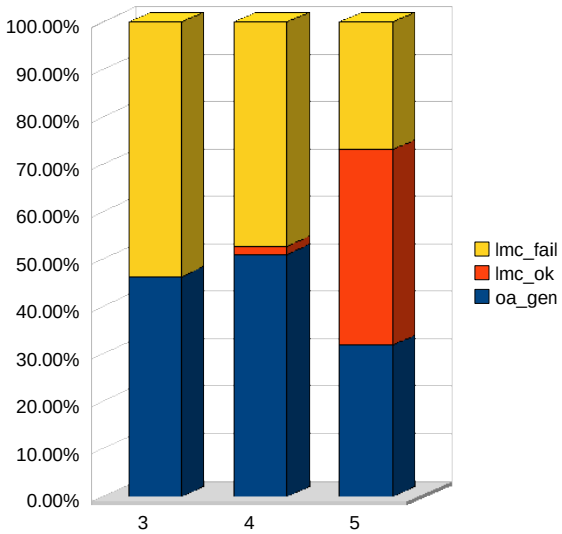
### 5.5.2 Multiple cpu breakdown

When combining the information from section 5.5.1 with the run times for more cpus and using the communication time model from section 5.2, it is possible to calculate the multiple cpu time breakdown. For the master and the slaves another class is introduced; waiting. For the master it is calculated by comparing the total time with the time needed for the generation of the arrays and the communication. For the slaves, the waiting is calculated by subtracting the communication and lmc test time from the total time. To compensate for the




| Column | Total time $[s]$ | Nr of arrays |
|---|---|---|
| 3 | 24 | 15680 |
| 4 | 153 | 1213 |
| 5 | 402 | 448 |
| 6 | 839 | 184 |

**Figure 5.2:** Calculation time break-down for $OA\left(27, 9 \cdot 3^5, 2\right)$




| Column | Total time $[s]$ | Nr of arrays |
|---|---|---|
| 3 | 125 | 14053664 |
| 4 | 70 | 3879920 |
| 5 | 1714 | 3914233 |

**Figure 5.3:** Calculation time break-down for $OA\left(27, 3^5, 2\right)$

number of processors, it is multiplied by the number of slaves used.
Figure 5.4 shows the results from a run with 4 cpus. The majority of the time is spent waiting, by the slaves as well as the master, `sl_wait` and `mst_wait`. It shows that the master as well as the slaves spend most of the time with other things than generating and testing arrays. This confirms the bad parallel speed-up found in section 5.3.

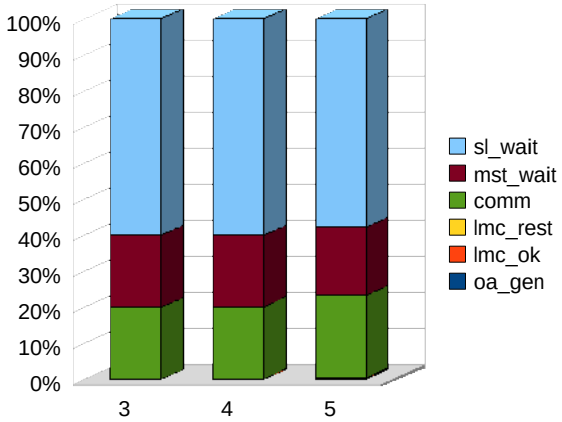
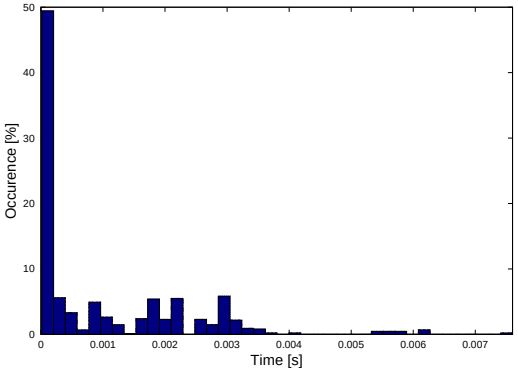


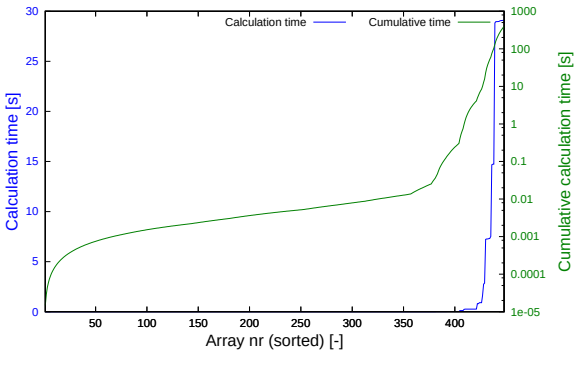

**Figure 5.4:** Multiple cpu breakdown for $OA\left(27, 9 \cdot 3^4, 2\right)$ with 4 cpus

## 5.6 LMC test time

In section 5.3 a bad speed-up is shown and in section 5.5 it is shown why this happens. When using multiple processors, the slaves as well as the master are waiting most of the time. The current system of parallelization makes the master prepare a task and let a slave handle it. With a higher number of processors in mind, it is favorable for the master to be ready before a new slave is, so a waiting master is no problem. But when the slaves are waiting, this indicates that the tasks are not prepared fast enough. The results shown in section 5.5 do not confirm this. The average time needed for an array is roughly one order of magnitude smaller than the average time needed for an lmc test. Therefore the time *per array* is measured in order to get the distribution of the time needed. Figure 5.5 shows the time distribution for the lmc test. The majority of the arrays needs far less time than the average value, which is higher due to a small minority of long runs.
Figure 5.5(a) shows a plot of the calculation time per array for a short run. It shows that half of all the runs are shorter than $t = 5 \cdot 10^{-4}\, s$ and only some of the runs take longer. For other columns in the array, the distribution is more extreme and a higher fraction is in the smallest bin. In figure 5.5(b) an example of a longer run is given; $t = 391\ s$. The time per array and the cumulative time of the sorted times are given. For the times per array, three types can be identified; arrays that did not pass the test in an early stage, arrays that did not make it in a later stage and arrays that passed the test. The first category does not contribute significantly to the total time but has the most arrays, about 2 %. The second category is smaller than the first but contributes roughly 20 % of the total time. The solutions are the smallest group, but take up the rest of the calculation time.
For figure 5.5(a) a bar plot could be used to indicate the distribution of the times per test.




(a) $OA\left(16, 2^{13}, 2\right)$, Column 6, $E\left(t\right) = 1.00\,ms$




(b) $OA\left(27, 9 \cdot 3^9, 2\right)$, Column 5, $E\left(t\right) = 0.879\,s$

**Figure 5.5:** Distribution of calculation time per lmc test

For other figures, almost all results would be in the smallest bin and the graph would be unreadable. All other results are shown like the graph in figure 5.5(b). It has the time per lmc test and the cumulative time up to that array.

# Chapter 6

# Discussion and Conclusion

An important conclusion is, that a fast program has been produced, which can be operated on a parallel network. The results listed in this research are characteristic for the hardware platform used. A cray supercomputer for example is optimized for communication purposes. It can easily be seen that communication intensive jobs run faster on such systems. For each calculation, parameters such as the variable type for the element of the arrays and the size of the counters can be varied as well. For different hard- and software platforms different settings are needed to get an optimal result.
Since the program written for this research was a replacement, it is useful to compare it with its predecessor. The comparisons show varying results, in some cases the previous program was faster. In other cases this program is much faster. In cases with a higher number of columns, this program is usually faster, with a lower number, the program by Snepvangers (2006) is usually a little bit faster. His program uses a depth first approach, so a per column comparison is not possible.
The program is designed to work on multiple processors, but does not succeed in gaining an acceleration from that. The opposite is encountered; more processors increase the run time. By looking at what the processors are doing, it can be seen that for both the master and his slaves, the majority of the time is spent waiting on each other. These results contradict each other; if the master spends most of its time waiting, it means it can generate arrays faster than the slaves can process them and *vice versa*. A waiting master suggests a system which can benefit from more processors. Since this is not the case, something else occurs. A reason for this contradiction could be in the communication of the processors; the benchmark was done under "ideal" conditions or the communication time model is not suitable in this situation. A more practical reason could be in the message queuing system of the computer used. If this system fails to process the communication between each processor efficiently, this could result in a slow down of the process.
In the beginning of the research, the lmc test was identified as the clear main bottleneck of the process. The number of permutations that an array has to go through is possibly high. Also there is no prior knowledge, so it was assumed that the time per array could not be predicted. A lot of effort is put into increasing the speed of lmc tests and trying to fail a lmc test as soon as possible. A high reduction has been realized, especially by the partial sorting using the columns from the root. By measuring the times per lmc test of an array, it can be seen that most arrays fail relatively early and only a small fraction takes up the majority of the time. From the timing results, it can be concluded that the effort put into the lmc test

has paid of.

Overall it can be said, that a all-round robust program has been produced. It performs well with one processor, but needs much improvement to take advantage of multiple processors. Initially, the lmc test was considered the most time consuming, since a solution needs so many permutations. And since no prior knowledge was available on how many permutations are needed for other arrays, this gained most attention for improvements. Now, measurements of times indicate that the bottleneck is the generation of the arrays. Due to its improvements and acceleration, some open cases have been calculated and one matches the prediction of the final number of solutions. This proves that the program is fit for its purpose and can be used to calculate unknown orthogonal arrays.

# Chapter 7

# Recommendations

Although this research has produced a valid and working program which meets its demands, improvements are possible. Especially for future use, a faster program can be desired for longer runs. These improvements can be made in the algorithm for the generation and testing of arrays, described in section 7.1. The implementation of these algorithms in the programming language can also be improved to generate a faster program. Section 7.2 gives recommendations on improvements on the code. The program is designed to work with the MPI library but with some alterations can be run on a condor network. The steps needed to achieve this are given in section 7.3.
When looking at the parallel speed-up of the program developed, some remarkable results are shown. When running with more processors, the program actually slows down. The reason for this odd behavior is found in the breakdown of the calculation time. The reduction in the calculation time of the lmc test has made this form of parallelization unfavorable. This can be improved (considerably) reducing the time for the generation of the arrays or by changing the distribution of the tasks.
In table 5.3 a remarkable case is shown with $OA\left(48, 4 \cdot 2^6, 3\right)$. By checking the breakdown of the tasks for this case, more insight can be gained in the drawbacks of the current program. A powerful combination can be made, by using the previous program for the starting solutions. The completion can be done by this program, since it is able to start with previous solutions. A small program which translates the results has to be created, but that is relatively easy.

## 7.1 Algorithm improvement

From the results it can be concluded that the current method of parallelization is not practical. The method itself can be changed or the generation of extensions can be accelerated. Currently, almost every value possible is tested and if it is allowed, it is used. An improvement of the generation of extensions can be by not just trying any value, but only values that are allowed. In the new column, the same values are only allowed a limited amount in the "blocks" of the partial sorting, mentioned in section 4.4. In column $k$ each value has to appear exactly $index \cdot \frac{s_t}{s_k}$ times. Different values can be obtained by permutations of this partial column. This ensures that only allowed values are used and simplifies the generation algorithm. Additional conditions, such as the first element is always 0, still have to be used to reduce the number of solutions possible.
When improving the extension generation is not enough, a more drastic improvement has to

be done. Snepvangers (2006) applies the parallelization on a different location in the algorithm; when multiple options are found for an element, the current state of the search tree is copied to another node. Both nodes continue with their own part of the search tree. This would reduce the amount of communication needed and the therefore would decrease the waiting time.

## 7.2 Code improvement

For the generation of the permutations of the sections and levels, a recursive function is used. It is a relatively small function, which gives the flexibility to be used with any setting for strength or levels. A drawback for a recursive function is the difficulty in the debugging process and the overhead of a function call. Therefore the recursive function should be replaced by a loop, which is also easier to optimize by compilers.
The lmc test has already been heavily optimized but for future uses, a faster test might be needed. The permutations needed during the test are generated each time they are needed and deleted afterwards. When an array with the same number of columns is being checked, the permutations from a previous call could be reused. This would add some acceleration, although care must be taken with the memory usage. Due to all the permutations, the amount of memory increases rapidly. Therefore a 64-bit compiler should be used when available.

## 7.3 Condor network adaption

When using the condor network, each computer is sent a batch job it is supposed to do and no communication is possible between processors. It could be possible to use our new program with the single processor option in this type of network. One processor or computer can calculate all the extensions of the root and save them. A small tool must be created which can read a result file and divide it into several smaller valid results files. They can be sent to other computers and they can extend this solution further. If the extensions yield a small number of solutions, a second round of collecting solutions files and distributing tasks might be needed.

# Bibliography


Addelman, S. and Kempthorne, O. (1961). Some main-effect plans and orthogonal arrays of strength two. *Annals of Mathematical Statistics*, pages 1167–1176.

Bisseling, R. H. (2004). *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, Oxford, UK.

Brouwer, A. (2008). Small mixed fractional factorial designs of strength 3. url-http://www.win.tue.nl/ aeb/codes/oa/.

Bulutoglu, D. and Margot, F. (2008). Classification of orthogonal arrays by integer programming. *Journal of Statistical Planning and Inference*, 138:654–666.

Bush, K. A. (1952). Orthogonal arrays of index unity. *Annals of Mathematical Statistics*, 23:426–434.

Delsarte, P. (1973). An algebraic approach to the association schemes of coding theory. *Philips Research Report Supplement*, 10.

Hedayat, A., Sloane, N., and Stufken, J. (1999). *Orthogonal Arrays, Theory and Applications*. Springer.

Rao, C. (1947a). Factorial experiments derivable from combinational arrangements of arrays. *Journal of Royal Statistical Society Series B*, 9:128–139.

Rao, C. R. (1947b). Factorial experiments derivable from combinatorial arrangements of array. *Journal of the Royal Statistical Society*, 9:128–139.

Schoen, E. D. and Nguyn, M. V. (2007). Enumeration and classification of orthogonal arrays. Unpublished Article.

Snepvangers, R. (2006). Statistical designs for high-throughput experimentation.

# Appendix A

# Program Usage

## A.1 Running options

For normal operations, the program only needs to be compiled once with a supplied makefile. The parameters for an orthogonal array can be set in the file `oaconfig.txt` in the following form:

`runs N`
`strength t`
`nfactors n`
$s_1$ $s_2$ $s_3$...$s_n$

First three lines with a variable and its value are given. The order is mandatory and *one* space is needed after the variable name. The list with the number of levels per column follows afterwards, all separated by one space. Any text that comes after the declaration of $s_n$ is not considered.

At the commandline several options are possible; the option `-h` or `--help` show a short help text with the commandline options. The level of output is controlled by the option `-l` followed by a number. In the file `tools.h` the different options are listed, but a value of 1 reduces the output to the essential lines. Higher values show more output and even debug information. Restarting can be controlled by `-r` followed by a filename containing the arrays. Proper values in `oaconfig.txt` are still needed for a correct operation.

## A.2 Compiling options

For normal usage, the program has to be compiled only once with the `Makefile` supplied. For some cases it could be necessary to change the size of a value in an orthogonal array. The standard data type is `int` but its value can vary with different systems and settings. Usually this is a 32 bit integer, but its length could also be 16 or 64 bits. Reducing the size of a value reduces the amount of memory needed and amount of data that needs to be transferred. The default setting is chosen, since an `int` is usually the standard data type for a processor and some speed-up can be obtained from that as well. The size of a single cell is defined by `array_t` and given in `tools.h`. The corresponding value of `MPI_ARRAY_T` should be changed as well below, to match the size of `array_t`.

# Appendix B

# Calculation Breakdown Details

This chapter gives the numbers used to make the figures in section 5.5. Table B.1 and B.2 give the numbers for the calculation time with one processor. Table B.3 contains the details of $OA\left(27, 9 \cdot 3^4, 2\right)$ when performed with four processors.

**Table B.1:** Calculation time details with one processor of $OA\left(27, 9 \cdot 3^5, 2\right)$

| Category | Tag | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| Failed lmc tests $[s]$ | `lmc_fail` | 0.12 | 0.04 | 0.04 | 0.02 |
| Passed lmc tests $[s]$ | `lmc_ok` | 22.03 | 129.79 | 303.86 | 631.90 |
| Generating extensions $[s]$ | `oa_gen` | 1.85 | 23.17 | 98.10 | 207.08 |
| Total $[s]$ | | 24 | 153 | 402 | 839 |

**Table B.2:** Calculation time details with one processor of $OA\left(27, 3^5, 2\right)$

| Category | Tag | 3 | 4 | 5 |
|---|---|---|---|---|
| Failed lmc tests $[s]$ | `lmc_fail` | 57.83 | 35.72 | 546.41 |
| Passed lmc tests $[s]$ | `lmc_ok` | 0.01 | 1.22 | 707.25 |
| Generating extensions $[s]$ | `oa_gen` | 67.16 | 33.06 | 460.34 |
| Total $[s]$ | | 125 | 70 | 1714 |

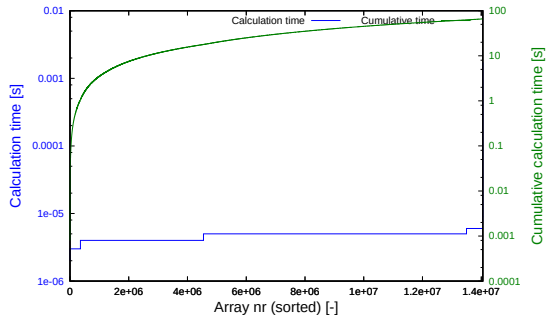
**Table B.3:** Calculation time details with four processors of $OA\left(27, 9 \cdot 3^4, 2\right)$

| Category | Tag | 3 | 4 | 5 |
|---|---|---|---|---|
| Failed lmc tests $[s]$ | `lmc_fail` | 0.12 | 0.04 | 0.04 |
| Passed lmc tests $[s]$ | `lmc_ok` | 22.03 | 129.79 | 303.86 |
| Generating extensions $[s]$ | `oa_gen` | 1.85 | 23.17 | 98.10 |
| Communication $[s]$ | `comm` | 2.25 | 0.13 | $2.39 \cdot 10^{-2}$ |
| Master waiting $[s]$ | `mst_wait` | 32.63 | 379.83 | 4032.94 |
| Slave waiting $[s]$ | `sl_wait` | 26.60 | 680.73 | 10893.05 |
| Total $[s]$ | | 35 | 380 | 4033 |

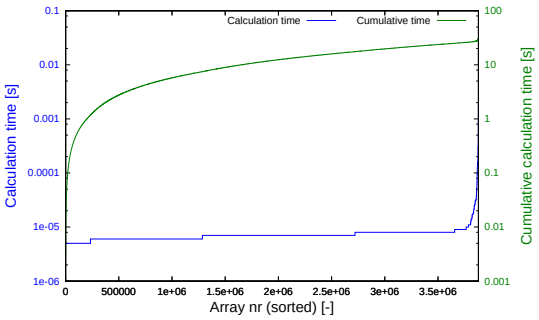
# Appendix C

# LMC Test Time Details

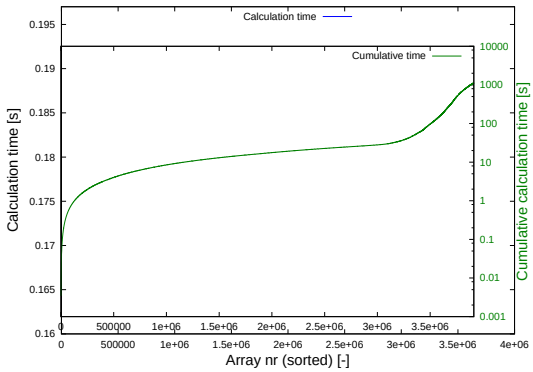
This chapter gives all the figures on the calculation time distribution of the lmc test. For both $OA\left(27, 3^6, 2\right)$ and $OA\left(27, 9 \cdot 3^5, 2\right)$ the details are given. It should be noted that for the second case two logarithmic axes were needed to display the data correctly. The mean value mentioned is the total time over the number of arrays generated. This indicates the difference between the mean value and the distribution shown.




(a) Column 3, $E\left(t\right) = 5\,\mu s$

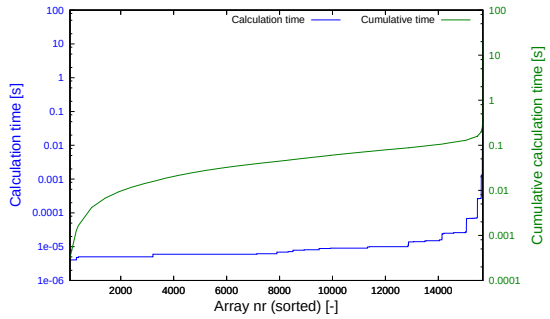



(b) Column 4, $E\left(t\right) = 8\,\mu s$

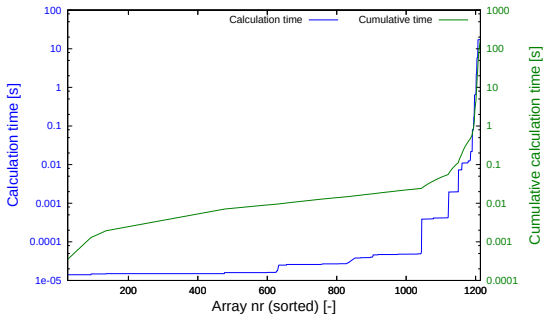



(c) Column 5, $E\left(t\right) = 289\,\mu s$

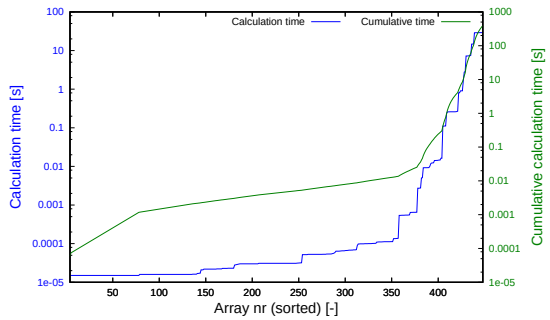
**Figure C.1:** Calculation details of lmc test for $OA\left(27, 3^6, 2\right)$



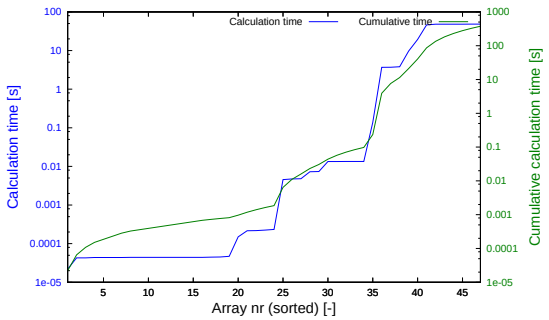

(a) Column 3, $E(t) = 1.47\,ms$




(b) Column 4, $E(t) = 0.13\,s$




(c) Column 5, $E(t) = 0.88\,s$




(d) Column 6, $E(t) = 8.00\,s$

**Figure C.2:** Calculation details of lmc test for $OA\left(27, 9 \cdot 3^5, 2\right)$