Parallel Algorithms for Sparse Grids in X10

Thomas Hörmann
Bachelor Thesis in Informatik

Parallel Algorithms for Sparse Grids in X10

Parallele Algorithmen für dünne Gitter in X10

Author: Thomas Hörmann
Supervisor: Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor: Dipl.-Inf. Martin Schreiber
          Benjamin Peherstorfer, M.Sc.
Date: June 17, 2013
I assure the single handed composition of this bachelor thesis only supported by declared resources.

München, den 17. Juni 2013

Thomas Hörmann
Abstract

The efficient integration of high dimensional functions is still an unsolved problem. For standard approaches we have an exponential dependency of the number of grid points. This is called the curse of dimensions. By using sparse grids we can overcome this to some extent. In this thesis we discuss how to integrate those functions by hierarchizing sparse grids. In order to solve them we show selected parallel algorithms for sparse grids. They are realized in X10, a programming language with implicit features concerning concurrency and distribution.
# Contents

## Abstract

### Outline of the Thesis

## I. Introduction and Theory

1. Sparse Grids
   1.1. Interpolation
   1.2. Integration
   1.3. Integration on High Dimensional Functions
   1.4. Hierarchical Base
   1.5. Sparse Grids

## II. Algorithms

4. Sequential Algorithms
   4.1. Recursive Algorithm
   4.2. A perfect hash function for regular sparse grids: gp2idx
   4.3. Iterative Algorithm
## Contents

5. **Parallel Algorithms** 33
   5.1. Algorithm for Shared Memory ................................. 33
   5.2. Algorithm for Distributed Memory 1: Distributed Sparse Grid 34
   5.3. Algorithm for Distributed Memory 2: Broadcast Computations 35
   5.4. Algorithm for Distributed Memory 3: Grid-Splitting .............. 36

III. **Results and Conclusion** 39

6. **Results** 41
   6.1. Test Environment ............................................. 41
   6.2. Sequential Algorithm ........................................... 42
   6.3. Algorithm for Shared Memory ................................ 42
   6.4. Algorithm for Distributed Memory 1 .......................... 43
   6.5. Algorithm for Distributed Memory 2 .......................... 44
   6.6. Algorithm for Distributed Memory 3 .......................... 45
   6.7. Comparison of the Algorithms ................................. 45
   6.8. Application .................................................... 46

7. **Conclusion** 47

**Appendix** 51

A. X10 Implementation for the second Distributed Memory Sparse Grid Algorithm 51

**Bibliography** 59
Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: SPARSE GRIDS
This chapter presents some principles of numerical interpolation and integration. We will transform the method to a hierarchical structure and create sparse grids as a result of the optimization of a cost-benefit equation.

CHAPTER 2: PARALLELISM
In this chapter we will discuss why we need parallelism. Furthermore we will explain some common architectures for parallel systems. We finish this chapter with a look on how we measure our algorithms and why these measurements are needed.

CHAPTER 3: X10
In this chapter we present the programming language X10. We will also explain some of its language features and classes which are provided to create parallel programs.

Part II: Algorithms

CHAPTER 4: SEQUENTIAL ALGORITHMS
This chapter presents some sequential algorithms that can hierarchize sparse grids. Furthermore we show data structures to store the sparse grid.

CHAPTER 5: PARALLEL ALGORITHMS
In this chapter we are introduce parallel algorithms for sparse grids. The algorithms are created by different purposes concerning the available hardware components and the size of the sparse grid.

Part III: Results and Conclusion

CHAPTER 6: RESULTS
This chapter presents the results of the algorithms introduced in Part II. We show some runtime measurements and also analyse the advantages and disadvantages of those algorithms.

CHAPTER 7: CONCLUSION
This chapter recapulates the thesis and compares the algorithms for their practical application.
Part I.

Introduction and Theory
1. Sparse Grids

In order to explain the world, modern science use models. They are simple abstractions of a complex system\cite{8}. Therefore models reduce complexity by ignoring less significant details. Most of them describe their correlations in a mathematical way. We obtain for example a partial differential equation. As a solver we can for example use the finite difference method. For higher dimensions this method has a crucial drawback. To increase the accuracy we need to evaluate much more points in our domain. The number of grid points increases by the power of the dimension in an order of $O(N^D)$ while the error decreases by $O(N^{-2})$. This is called the curse of dimensionality. Sparse grids as another solver can overcome this curse to some extent, under the condition that the function is sufficient smooth. Sparse grids work with $O(N \cdot \log N^{D-1})$ gird points with an error of $O(N^{-2} \cdot \log N^{D-1})$\cite{6}.

In the following we will describe some principles of interpolation and integration of one and more dimensional vector spaces. Afterwards we create a hierarchical basis, where sparse grids are the result of a work benefit optimizing.

1.1. Interpolation

The problem of generating a function $p : \mathbb{R} \to \mathbb{R}$ based on a set of $N$ points $(x_i, y_i), 0 \leq i < N$ from an unknown continuous function $f \in C^0$ is called interpolation. The function $p$ should at least match the condition $p(x_i) = y_i, \forall i.0 \leq i < N$. Since we assume a finite set of finite points our domain is bounded. Therefore we can restrict our domain to $x \in [0, 1]$ without loss of generality. One standard approach is to build the function $p$ by some base functions $\phi_i$ with the property

$$\phi_i(x_j) = \delta_{ij}, \forall i.0 \leq i, j < N,$$

(1.1)

where $\delta_{ij}$ stands for the kronecker delta

$$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}.$$

(1.2)

Therefore the function $p$ easily can be constructed as

$$p(x) = \sum_{i=0}^{n} y_i \phi_i(x).$$

(1.3)

The Lagrange polynomials represent a set of functions that satisfies the conditions of formula\cite{1.4}. 

\cite{90x672}
1. Sparse Grids

Figure 1.1.: A simple interpolation problem, known as drawing by numbers. On the left is the given problem by seven points. The point on the upper left counts twice as start and finish point. In the middle is the result for the Lagrange polynomial solver. On the right is the solution for piecewise linear functions.

\[ L_i(x) = \prod_{j=0 \atop i \neq j}^{n} \frac{x - x_j}{x_i - x_j} \] (1.4)

Using the Lagrange polynomials for interpolation will come with some disadvantages. For the evaluation of the polynomial we need an order of \(O(n)\) and an additional \(O(n^2)\) in preprocessing. Also the function gets instable for large \(n\) \([10]\). This can be seen in Figure 1.1 were \(n = 7\).

Another approach is to split our domain into smaller intervals with constant distance \(h = x_{i+1} - x_i\). The function gets also divided into piecewise polynomial of lower degree. This method is called spline interpolation. The interval is given by \(I_i = [x_i, x_{i+1}]\). This leads us to the piecewise defined linear function

\[ p(x) = \begin{cases} p_0(x), & x \in I_0 \\ \vdots \\ p_{n-1}(x), & x \in I_{n-1} \end{cases} \] (1.5)

where \(p_i\) is defined as

\[ p_i(x) = \frac{y_i - y_{i+1}}{x_i - x_{i+1}} + \frac{y_{i+1}x_i - y_ix_{i+1}}{x_i - x_{i+1}}. \] (1.6)

Figure 1.2.: Transfering piecewise linear functions into a base of hat functions.

This can easily be transformed to a problem written as a weighted sum of our base functions \(\varphi_i\) as in (1.3). The transformation is illustrated on Figure 1.2. In order to create a set of base functions we use the so called hat function. The hat function \(h_i\) is given by

\[ h_i(x) = \max \left\{ 1 - \left| \frac{x + x_i}{h} \right|, 0 \right\}. \] (1.7)
1.2. Integration

The integration of an unknown function can be very difficult. Since we know that we cannot even integrate every known function into a closed mathematical form, like the Gaussian Kernel Function, we also need a convergent numerical approach. The following idea is to interpolate the unknown function \( f \), and integrate this new function \( p \). We can see that the integral of the base function is independent of the interpolation points. So we can integrate the base function in advance.

\[
\int_0^1 p(x)dx = \int_0^1 \sum_{i=0}^{n} y_i L_i(x)dx = \sum_{i=0}^{n} y_i \int_a^b L_i(x)dx = \sum_{i=0}^{n} y_i \omega_i \tag{1.8}
\]

The integration based on the Lagrange polynomials using piecewise constant distances between the evaluation points is only stable for \( n < 7 \). We get negative weights for \( n \geq 7 \)\[10\]. A certain improvement of this method is called Gaussian quadrature. It chooses special positions for the evaluating points. This leads to a stable algorithm and increases the degree of polynomials that can be integrated exactly. The Gaussian quadrature works quite well for one dimensional problems, but it needs a lot of evaluation points on a higher dimension.

In the numerical integration we can also find some advantages in the second interpolation approach. The hat function can easily be integrated. They have all the same integrand except of the two at the boundaries. Denote that \( \omega_0 = \omega_n = \frac{h}{2} \) and \( \omega_i = \omega = h \) for all \( 0 < i < n \).

\[
\int_0^1 s(x)dx = \int_0^1 \sum_{i=0}^{n} y_i h_i(x)dx = \sum_{i=0}^{n} y_i \int_0^1 h_i(x)dx = \sum_{i=0}^{n} y_i \omega_i = \frac{\omega}{2}(y_0+y_n)+\omega \sum_{i=1}^{n-1} y_i \tag{1.9}
\]

1.3. Integration on High Dimensional Functions

In practical appliance most functions depend on more than just one variable. For example, we want to model a mechanical system in our three dimensional space. The time can also be an additional degree of freedom. We are not restricted to any maximum of dimensions, as applied in the field of Data Mining\[7\].

To represent a \( D \)-dimensional vector we use the following notation: \( \vec{x} = (x_1, ..., x_D) \). As in the one dimensional case, we restrict our domain to \([0, 1]^D\). For the base function we now use the multidimensional hat function called pagoda function\[6\].

\[
\varphi_i(\vec{x}) = \prod_{d=1}^{D} \max \left\{ 1 - \left| \frac{x_d + x_{id}}{h_d} \right|, 0 \right\}. \tag{1.10}
\]

fulfills the condition (1.1). By inserting more points into the domain we are able to increase accuracy.
1. Sparse Grids

Where $\vec{i}$ is the multidimensional index vector representing the position in the full grid given by the formula $x_i = i_i/N$. The interpolating function $p$ can be written as

$$p(\vec{x}) = \sum_{\vec{i} \in I} y_{\vec{i}} \varphi_{\vec{i}}(\vec{x}).$$

(1.11)

We can also easily compute the integral of our base function

$$\int_{\Omega} \varphi_{\vec{i}}(\vec{x}) d\vec{x} = \prod_{d=1}^{D} h_d \delta_{\vec{x}} = h^D. \quad (1.12)$$

The multidimensional integrand is given by

$$\int_{\Omega} s(\vec{x}) d\vec{x} = h^D \sum_{\vec{i} \in I} y_{\vec{i}}. \quad (1.13)$$

This representation shows a very simple algorithm that is perfectly well suited for parallelism. In contrary it leads to a very huge set of grid points. The grid points increase in an order of $O(N^D)$. Here we have the exponential dependency of the number of grid points again.

In order to overcome this complexity we have to use only a subset of the full grid. One approach is given by randomly choose some of the points in the full grid. By doing so, we also have to adapt the weight to $1/n$ where $n$ is the number of points selected from the full grid. This method is called monte carlo quadrature. It is a common way to compute the integral of a high dimensional function. It also provides a high scalability and its convergence is given by the law of great numbers.

1.4. Hierarchical Base

In order to increase accuracy, we have to gain more evaluation points. The previous representation of the function is not convenient for such an expansion. We cannot reuse the old structure of the grid and in the worst case scenario we are not able to use any of the evaluation points of the old grid. We are going to create a more dynamic structure. The goal will be, to add some new grid points without changing the previous structure or any of its coefficients.

At first, we create a hierarchical base for the one dimensional space. Afterwards we multiply successively this one dimensional space to our lower dimensional space in order
to reach higher dimensions. We will call the coefficients of this hierarchical base $\omega_{i,l}$, where $l$ is the level where the point was added, and $i$ is the index of the full grid with $2^l$ grid points. Therefore the position of the coefficient can be computed by $x_{i,l} = i \cdot h_l$. In case of the special domain $[0, 1]^D$ the mesh width $h_l$ can be computed as $h_l = \frac{1}{2^l}$. We are starting with the first level of our hierarchical base. This base just contains one point and hence is equivalent to the one point base of the full grid. Since this is the first point value of $\omega_{1,1}$ it is the same as the evaluation at this point $y_{1,1}$.

Figure 1.4.: The left side shows the base functions of the full grid. The right side shows the base functions for the hierarchical grid.

In the next step, we add $2^{l-1}$ points. The additional two points get the base function of the next full grid representation, but the old one stays the same. Now the base function also depends on the level.

$$\varphi_{i,l}(x) = \max \left\{ 1 - \left| \frac{x + x_{i,l}}{h_l} \right|, 0 \right\}.$$  \hspace{1cm} (1.14)

So the previous base contributes already a part of the result of the new point. The new coefficient depends on the old representation. For the second level they are computed as $\omega_{2,2} = y_{1,2} - \frac{y_{1,1}}{2}$ and $\omega_{3,2} = y_{3,2} - \frac{y_{1,1}}{2}$. This difference is called the hierarchical surplus. Denote that we suppose that our function is zero bounded. We get the more general formula to calculate the hierarchical surplus as followed.

$$\omega_{i,l} = y_{i,l} - \frac{y_{i-1,l} + y_{i+1,l}}{2}$$  \hspace{1cm} (1.15)

Now we make the step into the multidimensional base. This happens by multiplying the one dimensional hierarchical base to the $D - 1$ dimensional base.

$$\varphi((i_1, ..., i_D), (l_1, ..., l_{D-1}))(x_1, ..., x_D) = \varphi((i_1, ..., i_{D-1}), (l_1, ..., l_{D-1}))(x_1, ..., x_{D-1}) \cdot \varphi_{i_D, l_D}(x_D)$$  \hspace{1cm} (1.16)

For the multidimensional base we get this closed formula.
1. Sparse Grids

Figure 1.5.: Two dimensional hierarchical base [6].

\[
\varphi_{i,l}(\vec{x}) = \prod_{d=1}^{D} \varphi_{i_d,l_d}(x_d).
\] (1.17)

So we can compute its integral as followed.

\[
\int_{\Omega} \varphi_{i,l}(\vec{x}) d\vec{x} = \prod_{d=1}^{D} h_{l_d} \frac{1}{2^\|l\|_1},
\] (1.18)

where \(\| \cdot \|_1\) is the 1 norm on vectors

\[
\|l\|_1 = \sum_{i=1}^{n} |l_i|.
\] (1.19)

The calculation of the multidimensional coefficients is slightly more difficult, because we have to consider all the base functions where the overall level is smaller than its own level.

\[
\omega_{i,l} = y_{i,l} - \sum_{\tilde{l}, \|\tilde{l}\|_1 < \|l\|_1} \omega_{i,\tilde{l}} \varphi_{i,\tilde{l}}(x_{i,l}).
\] (1.20)

By using this formula to calculate the coefficients the computation gets more expensive for higher levels. However, we can reduce the dependencies to the \(2 \cdot D\) direct hierarchical parents as explained in [16] and also in section 4.1.

1.5. Sparse Grids

By restructuring the problem we neither reduce the error, nor we get any cheaper results. However an interesting effect can be noticed. The weight of the base function decreases by increasing the cumulative level. In particular, the sum of all weights of every subspace has
1.5. Sparse Grids

![Sparse Grid Diagram]

Figure 1.6.: Two dimensional base for sparse grid. For comparison the additional points for the full grid are in grey. It can be seen that even for small problems the work can be reduced\[6\].

the same value. By looking only at the base functions we can already see that all subspaces contribute as much as the others to the whole result. This leads to the approach to ignore subspaces with many points rather than subspaces with little nodes. This is quite a good method to reduce work.

For identifying which subspaces can be reduced we create a very simple cost-benefit equation\[6\]. For the costs of each subspace \( c(\bar{l}) \) we count the grid points and the benefit \( b(\bar{l}) \) is given by the weight of the base functions. So we get

\[
c(\bar{l}) = 2^{||\bar{l}||_1} \tag{1.21}
\]

for the cost function and

\[
b(\bar{l}) = c_b \tag{1.22}
\]

for the benefit function. We now define a global cost and a global benefit function\[6\].

\[
C(I) = \sum_{\bar{l} \in I} c(\bar{l}) \tag{1.23}
\]

\[
B(I) = \sum_{\bar{l} \in I} b(\bar{l}) \tag{1.24}
\]

We are now going to find an optimal solution for \( I \) by minimizing the global costs under the condition that the benefits are constant.

\[
\min_{B(I) = \text{const}} C(I) \tag{1.25}
\]

We now can use the simplification of our benefit function to find a solution. Since every subspace provides the same benefit \( b(\bar{l}) = c_b \) we can transform the optimization problem.

\[
\min_{|I| = \text{const}} C(I) \tag{1.26}
\]
1. Sparse Grids

For this, we can use a greedy algorithm where we only choose some $\vec{l}$ with a $c(\vec{l})$ value smaller or equal than a given constant. By choosing the subspaces by the criterion $c(\vec{l}) \leq const$ we get the conclusion $\|l\| \leq const$. In comparison to the full grid we can reduce the number of evaluation points massively to an order of $O(2^l \cdot l^{d-1})$ in comparison to the full grid with $O(2^d)$. The number of grid points is almost independent of the dimension. This can already be shown on very small grids as in Figure [1.6]

The same result can be achieved if we also take the values of the coefficients into account [6]. Furthermore we are able to estimate the error of our grid. We get an error of $O(2^{-2l} \cdot l^{d-1})$ by using the $L_2$ or the $L_\infty$ norm and an order of $O(2^{-l})$ by using energy norm $L_E$. 
2. Parallelism

The problems to be solved by using computers are getting bigger. This can be caused by different reasons. For example the model gets more detailed or the results are requested more accurate. So the execution time on the same machine is supposed to rise. To overcome this problem we either need some better algorithms or we have to use a system with a higher computation performance. For this there are two possible solutions. The first option is to use a faster machine. The second option is to use multiple devices of the same type. The clock rate has not increased radically since about the year 2000. This development is attributed to the increasing power consumption at rising clock rates[3]. For machines that need a very high computation power, only the second option is remaining. These systems can be classified by their memory access. We are going to look at two of them a bit closer. The first type is called shared memory system, the second type is called distributed memory systems.

2.1. Shared Memory System

In a shared memory system all processing units have access to the same address space. These may be the cores of a modern processor as well as several processors which are interconnected via a bus. Also the memory can be a combination of some smaller memory units. We can differentiate our system into two further sub groups. In the case that all processors can access the memory with the same bandwidth and latency, then it is called a uniform memory access (UMA). In the other case only the address space is shared. The machine consists of multiple memories and the access time depends on the location. This system is referred to as Non Uniform Memory Access (NUMA).

![Diagram of shared memory system](image)

Figure 2.1.: The diagram shows the basic structure of a shared memory system. All CPUs have direct access to memory via a bus. The figure on the left side shows a UMA system, where every CPU has the same access to the memory. The figure on the right shows a NUMA system. The access to the local memory is much faster than access to the remote memory.

The advantage of a shared memory system is the fast communication between the processors by just writing into the memory. All processes have access to the same data, which
2. Parallelism

2.2. Distributed Memory System

Current processors are limited in the number of cores. For example, the Intel Xeon Processor E7 family provides a maximum of 10\cite{ref15} and the AMD Opteron 6300 Series provides a maximum of 16\cite{ref1} cores per processor. In order to get even more computation power, we have to use multiple of these processors. These systems are often connected by a fast network\cite{ref20}. Since they do not share their memory to a global address space, they are called distributed memory system. Another reason to use a distributed memory system rather than a shared memory system can be the costs for the purchase\cite{ref3}. Especially on a large scale, distributed memory can be much cheaper than the shared memory equivalent. On a distributed memory system processes do not communicate directly through the memory, they send messages over the underlying network. Such a system can be arbitrarily expanded using additional processors or memory. However, the communication can have much higher latency.

2.3. Measurements

In this section it will be discussed what is being measured. We want to analyze the benefits that can be achieved by using multiple processors\cite{ref11}. The result for the achieved speedup is calculated by

\[
\text{speedup}(n \text{ cores}) = \frac{\text{execution time(1 core)}}{\text{execution time(n cores)}}
\]

This measurement is necessary because of an increase of the theoretical peak performance does not result in a better performance of every algorithm. This strongly depends on the algorithm itself. For example, the Monte Carlo Quadrature scales well by using many cores. Some other algorithms do not scale as well as the Monte Carlo Quadrature, like the algorithms explained in chapter 5 to compute sparse grids. This can be explained by Amdahl’s law\cite{ref2} to some extent. We divide the execution time of one core into a part that can be perfectly parallelized and in an inherently sequential part.

\[
\text{execution time(1 core)} = \text{sequential} + \text{parallel}
\]
2.3. Measurements

Figure 2.3.: The left graph shows the optimal trend of scalability. The computation power increases linear to the number of processors. The graph in the middle shows the maximum speedup based on Amdahl’s law. The right graph shows the trend of an application where at some point a maximum is reached. Afterwards the speedup is decreasing and therefore the application gets more inefficient. This trend can be observed at some sparse grid algorithms explained in chapter 5.

The serial part consists, for example, that variables have to be initialized. This has to be made by each processor. Only the parallel part can be parallelized. The execution time with n processors can be represented as

\[ \text{execution time}(n \text{ cores}) = \text{sequential} + \frac{\text{parallel}}{n}. \]

For the speedup we obtain the following formula

\[ \text{speedup}(n \text{ cores}) = \frac{\text{sequential} + \frac{\text{parallel}}{n}}{\text{sequential} + \frac{\text{parallel}}{n}}. \]

Thus, we can specify an upper limit for the speedup

\[ \lim_{n \to \infty} \text{speedup}(n \text{ cores}) = \frac{\text{sequential} + \frac{\text{parallel}}{n}}{\text{sequential}}. \]

Figure 2.4.: The figure shows an unbalanced work distribution between the threads. By synchronizing the faster threads are set to idle after their work. This causes a loss of computation time.
2. Parallelism

However Amdahl’s law is only a simplified model. In reality there are still other interferences that can limit the achievable speedup. As an example regular synchronizations, illustrated in Figure 2.4, can cause a lot of idle time. So computation time can get lost. This can be caused by an unbalanced work distribution. The processes with smaller tasks have to wait for the slower ones. Also the high latency connection caused by a remote memory call or the limited bandwidth of the network can decrease our parallel execution time and therefore reduce the scalability. In contrast to Amdahl’s law Gustafson’s law\cite{13} say, that we can solve larger problems by using more processors. In Gustafsons model we also say that we have a sequential and a parallel part. Instead of analyzing a fixed problem size we assume that our problem size increases. Since the sequential part is constant for the application, we can solve the larger problem by using more processors. This law also says that for very large problems, the sequential part gets less significant. We can also observe this effect on some of our algorithms for sparse grid.
3. X10

X10 is a high level programming language first presented by IBM\[21\] in 2004. Its source code is published at Sourceforge.net\[9\], but the development is driven by IBM. So it was steadily enhanced to its current version 2.3.1, released in March 2013. In May 2011 the X10 development team announced that the language specification is considered to be fairly stable\[23\], so that code written for Version 2.2 or later will also run on future releases. X10 creates high performance assemblies by using a translation to C++ from the original X10 code. Despite to the C++ code to code compiler there is a X10 to Java compiler, too. There also is a direct compiler for creating assemblies from X10 code without intermediate language. However, this compiler is in an early stage of development\[4\]. The X10 runtime library is also provided for both languages, C++ and Java. We are going to use just the C++ backend, because native code achieves better results in most applications than Java assemblies\[12\]. X10 is created with the background of implicit distribution and concurrency. It is part of the partitioned global address space (PGAS) family. This means, it has a global virtual memory space where any process can use some direct or indirect mechanism to access the data of another process\[24\]. Since most of the serial features are well known from other languages like C++ and Java we will concentrate to the features concerning distribution and concurrency.

3.1. Language Features

Places and Activities are two of the central concepts of X10. The data is structured by using Places. A Place can be thought of as a virtual shared-memory multi-processor\[24\]. They are all created directly at the start. Every Place has a unique id beginning by 0. The application itself is divided into small tasks called Activities. Activities are dynamically generated and terminated during the runtime. An Activity can have different states, like running, blocked or terminated. They are responsible for processing the data. In contrast to other languages, the constructs for managing Activities and the data transfer via a virtual Place shifting of the Activities are an integral part of the language. In the following we will describe some language features to control the Activities.

3.1.1. async

At the beginning of the application X10 starts exact one Activity on Place 0 by invoking the main method. This Activity is called the root Activity. Every Activity is able to create new Activities. This can be done by using the keyword async. async S invokes that a new Activity is created and executes the statement S. The parent Activity continues with the code after S. An Activity can only terminate, if all its children and therefore all its descendants have terminated\[12\].
3. X10

```java
// Listing 1.1
public static def main(Array[ String ])
{
    async task1();
    task2();
    async task3();
}
```

Figure 3.1.: This figure shows the execution flow of the sample code given by Listing 1.1. Task1 runs parallel to task2. After task2 has finished task3 is started asynchronous. Since there are no more tasks left, the root Activity has to wait until all children (in this case task1 and task3) has terminated.

By creating a new Activity the X10 Runtime creates a small heap object, where the local data is stored. The local data contains all variables defined in the async block as well as some immutable variables defined outside of the statement but used inside the block. So the creation of new Activities is provided with some overhead. Therefore for many tiny tasks it can be practical to do them in a sequential way rather than in parallel.

```java
// Listing 1.2
val array: Array[ int ] = new Array[ int ](10);

// increment all values sequential
for (val i in array)
{
    array(i)++;
}

// increment all values parallel
for (val i in array) async{
    array(i)++;
}
```

3.1.2. finish

In order to make sure, that a certain task only continues if others has finished their work, we can use the keyword `finish`. By executing `finish` the Activity waits until every child created in this block is terminated. After that, the execution of the Activity continues.
3.1. Language Features

```java
// Listing 1.3
public static def main(Array[String]){
    finish{
        async task1();
        async task2();
    }
    finish async task3();
}
```

Figure 3.2.: This figure shows the execution flow of the sample code given by Listing 1.3. In this sample task1 and task2 run in parallel. The root Activity is forced to wait until both tasks has finished. After that, task3 is started on another Activity. Because of the finish async Task3 behaves like the sequential execution[19].

The concatenation of two or more finish blocks does not affect the execution flow. The only difference can be seen, if an exception is thrown. The runtime takes the thrown exception and wraps it into another x10.lang.MultipleExceptions exception[24].

3.1.3. at

An Activity can only access, beside the data on its local heap object, the data on the Place where it is running on. If the Activity wants to access remote data it has to switch the Place. The at operation is used for Place-shifting. Any Activity has the ability to switch from any Place to another. This can be used to distribute the data and the workload or just to communicate between the different Places. By using the at operation the local heap object gets copied to the other Place. This local data is called closure and gets identified at compile time[12]. The compiler checks what variables are used at the other Place and therefore only send them at runtime to the new Place. The closure is restricted to immutable variables. By the standard serialization the closure contains all objects which are used in the new Place and all the objects referenced by them. This could be a problem for example if you work on a large tree, but only need the root element on the other Place. Therefore X10 allows overwriting the serialization. After the at statement or expression the Activity returns to its original Place. For this, the serialization is called again.
```java
// Listing 1.4
public static def main(Array[String]) {
    // Part 1: Traverse manually
    at (here.next()) {
        at (here.next()) {
            task1();
        }
    }
    // Part 2: Traverse by the Iterator
    for (val p in Place.places()) at (p) {
        task(p.id);
    }
    // Part 3: Compute asynchronous on every Place
    finish for (val p in Place.places()) async at (p) {
        task(p.id);
    }
}
```

Figure 3.3.: This figure shows the execution flow of the sample code given by Listing 1.4. In the first part, the Activity statically traverses the first two Places. The Activity does not return to Place 0 until it has finished its task. In the second part, the Activity executes a task on every Place. After finishing, the Activity has to return to Place 0 in order to continue. The third part shows the asynchronous execution of all tasks on every Place. The creator Activity waits on Place 0 until all child Activities has finished.
3.1. Language Features

3.1.4. atomic

Sometimes a specific part of the code should not be processed by two or more Activities. This could be if different Activities have to write on the same data. So you have to protect this code not to be accessed by others while one Activity is processing. We call this exclusive used part critical. This can be done by making the code atomic. This can be thought of creating a new single processor instruction that can be used only once per time and Place. The atomic can be used for example to increment a counter or insert and delete elements of a stack or queue.

```java
// Listing 1.5
static val count:int = 0;
public static def main(Array[String]){  
    for(val i in (1..3)) async
        atomic count ++;
}
```

Figure 3.4.: This figure shows one possible execution flow of the sample code given by Listing 1.5. The increment of the variable count is protected by an atomic expression. Only one Activity can increment the variable at once. The outcome of count is ensured to be 3 at the end of the application[19].

The atomic block is realized by mutex. This means each atomic block has a lock and every atomic block has its own lock. If an Activity runs into this block it checks if it is free to enter. If yes, the Activity sets the mutex occupied, else it sets its own state to blocked and waits until a signal is sent, that the lock is open again. They only protect the access to a certain part of the code, not the data accessed in the atomic block. An example for unprotected data accessed in an atomic block is shown by listing 1.6. Also an atomic block has no effect to the access of the code in other places[24].

```java
// Listing 1.6
static val count:int = 0;
public static def main(Array[String]){  
    for(val i in (1..2)) async{
        atomic count ++; // increment 1
        atomic count ++; // increment 2
    }
}
```
3. X10

Figure 3.5: This figure shows the execution flow of the sample code given by Listing 1.6. Only the access to increment 1 and 2 are protected independently. The variable in the atomic block can also be modified outside of the atomic block. This can lead to race conditions. The outcome of this algorithm is not deterministic.

3.2. X10 Class Library

The class library of X10 provides a set of different classes that provide useful objects for distribution and concurrency. We will explain some of the parallel classes we will need for our sparse grid algorithms. A detailed description of the entire class library can be found in the Online Documentation for the X10 Standard Library.

3.2.1. DistArray

The DistArray is an array of some uniform data distributed over different Places. In contrary a normal Array is supposed to store all data at the same Place where it was created. The data can be a set of objects which only a reference to the object is stored or a struct like Integer or Double, where the whole information is stored in the array itself. The DistArray is created by a Dist object that identifies which range of the array shall be mapped to which Place. In the simplest case the range is divided by the number of existing Places and then ordered into the Places. If the Dist object is created by

\[ \text{Dist.makeConstant(Region r, Place p)} \]

the DistArray acts like a normal array. A constant Dist map all points to the same Place. In the following listing we create a new DistArray that divide all points into a fair distribution.

```
// Listing 1.7
val N = 17;
val da = DistArray.make[Double](Dist.makeBlock(0..(N-1)));
```

If the number of Places divides \( N \) every Place contains the same amount of elements of this DistArray. Otherwise the Places with a smaller id get one element more than the last one. The data of a specific cell of the DistArray can only be accessed if the Activity also is at the same Place where the data is stored. In order to find the right Place we can use the underlying Dist object.
3.2. X10 Class Library

Figure 3.6.: This figure shows the outcome of the sample code given by Listing 1.7. A DistArray was created for \(N=17\) and 3 Places. Since 3 do not divide 17 the remainder of 2 was filled into the first 2 places. So Place 0 and Place 1 contain 6 elements and Place 2 only contains 5 elements. The maximum difference between two local DistArrays is 1\(^{19}\).

```java
// Listing 1.8
val point:int = 9;
// Identifying the Place where the data is stored
val place:Place = da.dist(point);
// Read a value from the DistArray
val value:Double = at(place) da(point);
// Write into the DistArray
at(place) da(point) += 2;
```

To iterate over the DistArray it can be more practical to choose the Place first and then iterate over the local part. So we do not have to shift the Place for every element.

```java
// Listing 1.9
finish for(val p in Place.places()) async at(p){
    val localDistArray = da | p;
    for(val point in localDistArray){
        localDistArray(point) ++;
    }
}
```

Figure 3.7.: This figure shows the execution flow of the sample code given by Listing 1.9. The Activities create a local copy of the global DistArray. This new DistArray only contains elements of the local Place. So they do not have to shift the Place again. Now they can traverse every point in the array. For performance reasons, X10 do not copy the local array itself, it only creates a new reference to the existing array using the same data\(^{19}\).
3.2.2. GlobalRef

The `GlobalRef` object globally addresses a piece of data. The object only contains a reference of the object, so it can be copied very cheap between different Places. This can be used if the object references to a big object or the Activity has to change the Place very often. If the Activity has to access the referenced data it has to switch the Place to the home of this object. The Activity can also read the data by call the method `getLocalOrCopy()`.

3.2.3. RemoteArray

Very similar but more specific acts the object is the `RemoteArray` object. It only encapsulates the reference to an `Array` as the `GlobalRef` object does, but it also provides an interface to access the `Array` elements. Again, the elements of the referenced `Array` can only be accessed if the Activity is in the Place where the `Array` was created. `RemoteArray` is also used to address an `Array` for the `asyncCopy` call.

// Listing 1.10
static val array = new Array[Double](5);
public static def main(Array[String]){  
val ra:RemoteArray[Double] = new RemoteArray[Double](array);
  at (here.next()){
    Array.asyncCopy(array, 0, ra, 0, 5);
  }
}

Figure 3.8.: This figure shows one possible execution flow of the sample code given by Listing 1.11. Every increment is protected by the `AtomicInteger`. The outcome of `count` for this algorithms will always be 4. The order in which `count` will be incremented is not deterministic.

3.2.4. AtomicInteger

As mentioned before, the `atomic` block only protects the code being accessed by different Activities. If the data itself should be accessed only by one Activity at a time, X10 provides the `AtomicReference` class. This class ensures that the access to the reference can only be modified by one Activity at a time. The referenced object is still not protected
by AtomicReference. For some basic types X10 also has a special collection for atomic access. These types are: Boolean, Double, Float, Integer and Long. For them, we can also protect the data.

The AtomicInteger class provides basic atomic methods, like get and set, as the AtomicReference does. The AtomicInteger also include some arithmetic operations. We can easily create a counter by using getAndIncrement()

```java
// Listing 1.11
static val count: AtomicInteger = new AtomicInteger(0);
public static def main(Array[String]){
  for(val i in 1..2) async{
    count.getAndIncrement(); // increment 1
    count.getAndIncrement(); // increment 2
  }
```
3. X10
Part II.

Algorithms
4. Sequential Algorithms

We start with the sequential algorithm. It is needed as a reference for the speedup measurements. Also it was used to validate the results of the parallel algorithms. Furthermore the sequential algorithm shows some data dependencies of the sparse grids, which are important for parallelism.

4.1. Recursive Algorithm

Most of the hierarchical structures are defined recursively. Therefore it seems obvious to use a recursive function. In order to create sparse grids we have defined a one dimensional hierarchical structure in section 1.4. Afterwards we successively multiply this one dimensional structure to reach the desired dimension. We will also use this technique to create our algorithm. At first we just concentrate on the one dimensional case.

Our input data can be retrieved by the function \( f \) as defined in Listing 2.1. As an input the function gets the level and the index vector and returns the value of the given point. If necessary, the exact position can be easily calculated by \( x_{i,l} = i \cdot 0.5^l \). For other domains beside \([0; 1]^d\) one could easily transform the input position by linear scaling and shifting. At the beginning, we use \texttt{int} instead of an array of ints of size 1. For the multidimensional function \( f \) it is important, to use arrays in the signature.

```java
// Listing 2.1
// One dimensional case
def f(level: int, index: int): double;

// General case for any dimension
def f(level: Array[int], index: Array[int]): double;
```

![Figure 4.1.](image.png)

Figure 4.1.: This picture illustrates the call graph of a one dimensional grid.

To save the results we can use a binary tree.
4. Sequential Algorithms

// Listing 2.2
class BinTree{
    var right: BinTree;
    var left: BinTree;
    var value: double;
}

In the one dimensional grid this is all we need to start the hierarchization.

// Listing 2.3
def hierarchize(level: int, index: int, bt: BinTree, leftValue: double, rightValue: double){
    val value: double = f(level, index);
    bt.value = value - (rightValue + leftValue) / 2;
    if (level < maxLevel){
        // Create tree nodes for the next level and compute their values
        bt.left = new BinTree();
        bt.right = new BinTree();
        hierarchize(level + 1, index * 2 - 1, bt.left, leftValue, value);
        hierarchize(level + 1, index * 2 + 1, bt.right, value, rightValue);
    }
}

For the multidimensional traversal of the grid we need an additional parameter for the dimension. We also add a level parameter. We just could calculate the cumulated level by summing up the components of the level vector, but to save computation time we use it as a parameter. As mentioned before, instead of a single int for the level and index we now use an array of ints. In the recursion we first increase the dimension and afterwards we increase the level. By doing so, we make sure not to traverse any node twice. The bigger problem is, that every node has $2 \cdot D$ hierarchical parent nodes. For the computation of one point we need all of them.

Figure 4.2.: The left side shows the call graph for a two dimensional sparse grid. The right side shows the hierarchical parents of one given point.

---

28
A possible solution could be a new function that returns the desired point in some sense. This could be the value of the node, or a pointer to the node. It will start again at the root node and search top down\[14\]. Another solution could be to expand the tree node with a back reference to its parents. These solutions are quite inefficient for large trees.

```java
// Listing 2.4
def hierarchize(level: Array[int], index: Array[int], curLevel: int, curDimension: int, lock: Boolean): void{
    // Make sure we do not traverse any grid point twice
    if(curDimension < maxDimension - 1){
        hierarchize(level, index, curLevel, curDimension + 1, true);
    }
    // Only hierarchize until a predefined maxLevel is reached
    if(curLevel < maxLevel){
        level(curDimension)++;
        index(curDimension) = 2 * index(curDimension) - 1;
        hierarchize(level, index, curLevel + 1, curDimension, false);
        index(curDimension) += 2;
        hierarchize(level, index, curLevel + 1, curDimension, false);
        index(curDimension) /= 2;
        level(curDimension) --;
    }
    // The lock is set by the dimension traversal to ensure we
    // compute every grid point only once
    if(!lock){
        var value: double = getValue(level, index);
        for(d in 0..maxDimension - 1){
            value -= getLeftParentValue(level, index, d) * 0.5;
            value -= getRightParentValue(level, index, d) * 0.5;
        }
        setValue(level, index, value);
    }
}
```

To store the values of the nodes we also can use hash tables\[17\]. This enables us to access the grid points in almost constant time. The size of the key depends on the dimension of the sparse grid. Therefore we get an order of $O(d)$ for the access time to any grid point. By using hash tables we can reduce computation time\[16\]. The structure of the sparse grid is still given by the recursion calls. The key for the hash table shall be a combination of the level and index vector. Common hash tables also have some disadvantages. As in the trees, we do not have data locality. The hash function shall map every point to a different index in the memory. This is bad for the cache. By randomly choose an index for the hash array, there could be some collision between two disjoint points when the hash function map them to the same index. So we have to save also the key for the value of the grid point. This leads to a big memory overhead.
4. Sequential Algorithms

4.2. A perfect hash function for regular sparse grids: gp2idx

We will now define a hash function for sparse grids, that offers us a perfect mapping of every grid point of a regular sparse grid, as well as an inverse function that allows us to iterate very fast through the sparse grid. It has been shown that this data structure is more efficient than other data structures commonly used for sparse grids\cite{16}. Therefore we will use it for our further algorithms.

The first optimization of the hash function should be avoiding collisions. If there are no collisions, we do not have to save the key anymore. So we try to find an injective hash function for a given sparse grid. For the regular sparse grid it is possible to find a bijektiv hash function with optimal mapping. Sparse grids have some regularities, we can use for the hash function. For instance every subspace on the same cumulative level has the same amount of grid points. A subspace is only a full grid with different mesh sizes. The calculation of the index for any grid point can be done in three steps. These steps are illustrated by Figure 4.3. In the first step we have to identify the position of the grid point in its subspace (black). In the second step we take the subspaces of the same cumulative level and bring them into a total order (green). The third step counts all grid points with a lower level (blue).

For the first step we have to find the position of the grid point inside the subspace given by the level vector. The mesh width in each dimension is given by $2^{l_i}$. So it is the same problem as addressing a point in a multidimensional array.

\begin{verbatim}
// Listing 2.5
var index1 : int = 0;
for (var t:int = 0; t < dimension; t++){
    index1 <<= (level(t) - 1); // *= Math.pow(2, level(t) - 1)
    index1 += (index(t) >> 1); // += (index(t) - 1) / 2
}
\end{verbatim}

The second step contains to bring the subspaces of one level into a total order. This happens by analyzing the components of the vector. We count the possible combinations of partition the sum $\sum_{i=1}^{t} l_i$ of the subvector into the subvector and subtract the combinations by fixing the last component. This can be written as a simple loop.
4.3. Iterative Algorithm

By using this data structure we do not have to save the level and index vector anymore. Therefore we can save a lot of memory overhead. This hash function comes with some other advantages. Since the function is bijektiv, we can define an inverse \( \text{idx2gp} \). So we can create a level and index vector by using the index of the hash table. Also the \( \text{gp2idx} \) function is optimal, so we have no empty gaps in the hash table. This allows us to iterate through the table. For computing the values of the grid points, we only need other points with a lower cumulative level value. These parents all have a lower index than the computed grid point itself. So we can define a loop to compute our coefficients. As in the recursive algorithm, we have to compute the whole tree by dimension.

```java
// Listing 2.8
for (var d: int = 0; d < dimension; d++){
    for (var j: int=start; j > 0; j--){
        idx2gp(level, index, j);
        values(j) -= getLeftParentValue(level, index, d) * 0.5;
        values(j) -= getRightParentValue(level, index, d) * 0.5;
    }
}
```
4. Sequential Algorithms

Let us look at the function `getLeftParentValue` a bit closer, how to find the hierarchical parent.

```java
private static def getLeftParentValue(
    level : Array[int], index : Array[int],
    dimension : int, direction : int): double {

    // Check for boundaries
    if (index(dimension) == 1)
        return 0;

    // Save coordinates of the original grid point
    val tempLevel : int = level(dimension);
    val tempIndex : int = index(dimension);

    // Compute the hierarchical parent
    index(dimension) --;
    while (((index(dimension) & 1) == 0){
        index(dimension) >>= 1;
        level(dimension) --;
    }
    val idx = gp2idx(level, index);

    // Reset level and index vector
    level(dimension) = tempLevel;
    index(dimension) = tempIndex;

    // Return value of the hierarchical parent
    return values(idx);
}
```

At first we check whether the desired point is at the border or not. In case of a point at the border we can evaluate a grid, which is representing the border. Since we suppose that we have a zero bounded problem, we can return the value 0. In the next step, we save the coordinates of the given grid point. This is necessary, because we need this point for further computations. After that, we compute the hierarchical parent. This can be done by decreasing the level as long as we have an even index number. After we have found the parent node, we can return its value. Right before we return, we have to reset the level and index vectors.
5. Parallel Algorithms

The next task is to parallelize the algorithm. First, we focus on the parallelization of the algorithm for a shared memory system. Afterwards we extend it to work on a distributed memory system. For this, three different approaches were chosen. The first approach is to distribute the data and the work evenly across the entire memory. In the second approach only the work is distributed. Thus, the access time shall be reduced at the expense of memory. The last approach takes a different view. Here, the sparse grid is split into smaller sparse grids.

5.1. Algorithm for Shared Memory

For this algorithm, we will limit ourselves to shared memory systems. So we can focus on the parallelization, since we do not have to worry about the layout of the memory. At first, we only concentrate on the hierarchization. Let us look at the sequential algorithm again. The sequential algorithm iterates backwards through the hash table. This is because the calculation of a grid point is independent from all points with a higher index. Next, we can say that we only need the hierarchical parent nodes for the computation. These all have a smaller value of their cumulative level. From this we can infer that all the elements on the same level are independent. Therefore we can process all elements with the same level in parallel. Since the elements are dependent on higher levels of those with lower levels we must ensure that the next level will not be started until the previous one has been fully completed. Thus we obtain the following code for the main loop of the algorithm.

```java
// Listing 2.10
for (val dim in (0..(dimension-1))){
    for (var l:int=maxLevel; l>0; l--){
        finish for (val thread in (0..(numThreads-1))) async{
            val index: Array[int] = new Array[int](dimension);
            val level: Array[int] = new Array[int](dimension);
            val range: Region = range(index3Lookup(l-1), index3Lookup(l),
                                      thread, numThreads);

            for (val idx in range){
                idx2gp(level, index, idx);
                val lParent:double = parent(level, index, dim, -1);
                val rParent:double = parent(level, index, dim, 1);
                values(idx) -= (lParent + rParent) * 0.5;
            }
        }
    }
}
```
5. Parallel Algorithms

5.2. Algorithm for Distributed Memory 1: Distributed Sparse Grid

The first distributed algorithm follows a fairly simple goal. We distribute the memory of the sparse grid all over the memory of all machines. At the same time we determine which place has to compute which part of the sparse grid. The advantage of this algorithm is that the memory of all machines is optimally utilized. The dependencies of a grid point to its parent node is also easy to handle, because the index structure is preserved by the DistArray. Each grid point can be accessed by using the place shifting. This is done by the keyword at as shown in section 3.1.3. In a uniform cost model, this approach should scale well. In reality, access to a remote memory takes much longer than access the local memory. This is due to latency of the network and the time needed for serialization. So we can definitely expect performance losses. As a consequence, we would at least restrict write access to the local machine. Every place computes the grid points that are stored locally. For this we have to modify the hash function as well as the hash table. Since all elements of the same level are independent of each other, they should be divided. Therefore we have to distribute our values array by level. The modifications are based on the index3 value of the gp2idx function defined in section 4.2. The index3 organizes the grid points by their cumulative level. So we have to split up the hash table at those points. We create a two dimensional array. The first dimension arranges the elements according to their level and the second orders the elements by using the gp2idx structure. We get the following code to create our values array.

```java
// Listing 2.11
val values = new Array[DistArray[Double]](maxLevel);
for (var i:int=0; i<maxLevel; i++) {
    var add:int = binomial(dimension−1+i, dimension−1) << i;
    values(i) = DistArray.make[Double](Dist.makeBlock(0..(add−1)));
}
```

Figure 5.1.: This figure shows a gp2idx hash table devided by level. Then the smaller arrays can easily be distributed over the different Places.

This gives us an almost uniform distribution of our grid points. We can use the distribution that we obtain by the DistArrays to divide our workload. Every Place shall only compute the values of the local array. For the main loop, we get the following code.
5.3. Algorithm for Distributed Memory 2: Broadcast Computations

// Listing 2.12
for (val dim: int in (0..dimension - 1)){
    for (var i:int = maxLevel - 1; i >= 0; i--){
        val l = i;
        finish for (val p in Place.places()) async at (p){
            val valuesP = values(l) | p;
            val index: Array[int] = new Array[int](dimension);
            val level: Array[int] = new Array[int](dimension);
            for (val point in valuesP){
                idx2gp(level, index, point(0) + offset);
                val lParent: double = parent(level, index, dim, -1);
                val rParent: double = parent(level, index, dim, 1);
                values(l)(point) -= (lParent + rParent) * 0.5;
            }
        }
    }
}

Per iteration, only one write access takes place. As we iterate only through the local array, it is ensured that the access takes place locally. Read access takes place in the parent function. We still need to make sure that this happens in the right Place. To reduce the overhead caused by serialization we first check if we are already at the right Place.

// Listing 2.13
if (values(levelSum).dist(idx) == here){
    return values(levelSum)(idx);
} else{
    return at(values(levelSum).dist(idx)) values(levelSum)(idx);
}

As another improvement we can adjust the functions gp2idx and idx2gp, that they do not need the index anymore.

5.3. Algorithm for Distributed Memory 2: Broadcast Computations

In the second algorithm we want to avoid the many remote accesses. All the data should be available on every machine. Therefore we have a fast access to the grid points. For this, these must always update the computed data after every step. The update should be as seldom as possible. This happens at the end of each dimension. For this, we need to change only the main loop.

The advantages of this algorithm are that we have a fast access to all values. Purely from the hierarchisation of the sparse grid, we can gain a speedup. However, we need to copy the computed data. This is clearly a more of work compared to the shared memory algorithm. We also can see that we get more work by using more places. The additional overhead is created by copying. By large grids we have to send much data. So latency
5. Parallel Algorithms

is only a small part of the overhead. Hence, the bottleneck here is the bandwidth of the underlying network.

For the realization of this algorithm, we need a way to address the remote arrays. For this we use RemoteArray.

// Listing 2.14
val remoteArrays = new Array[RemoteArray](Place.MAX PLACES);
// Create Table of all local value arrays
for (val pl in Place.places()){
    remoteArrays(pl.id)= at (pl) new RemoteArray(values);
}
// Send Table to all Places
for (val pl in Place.places()){
    remoteValues = remoteArrays;
}

The distribution of the work is done in the same manner as in the first distributed algorithm. At first we partition our values array by level. Then we distributed these parts to the places. Finally we divide them among the threads. After each dimension we have to broadcast the results.

// Listing 2.15
if (threadStop−threadStart >0){
    for (val pl in Place.places()){
        if (pl!=here)
            Array.asyncCopy(values(1), threadStart, remoteValues(pl.id),
                          threadStart, threadStop−threadStart);
    }
}

5.4. Algorithm for Distributed Memory 3: Grid-Splitting

The third algorithm chooses a completely different way. Here the parallelization is not done by distributing the work of one sparse grid to many cores. The distribution is done by creating many smaller sparse grids from the original one. For this we go back once again to the recursive algorithm. We traversed the sparse grid recursively first by dimension, than by level. Since we have here, as usual for a recursion, running the same code, we get on each step three new sparse grids. Two of them are smaller in the level. The third one has the same level depth, but is reduced in the dimension. This is illustrated Figure 5.2. So we can create smaller grids. These grids can then computed by the shared memory algorithm. Furthermore we need an algorithm that implicit support non zero bounded sparse grids. The edges can easily be represented by other sparse grids with the dimension reduced by one.

The advantage of this algorithm is that there are no dependencies between the grids. Each of these grids can be computed independently. Then we can use the shared memory algorithm to compute the smaller grids. This algorithm has also a drawback. We have to compute the edges of the smaller grids twice. This leads to a memory overhead and
5.4. Algorithm for Distributed Memory 3: Grid-Splitting

Figure 5.2.: This figure shows a three dimensional sparse grid. This grid is split into three smaller sparse grids (green, red and blue). The green and blue grids need also some points of the red grid for their evaluation. Because of dependencies of the green and the blue grid to the red one, only the red grid can be split into another three smaller grids.

greater overall work. This algorithm comes also with some restrictions. We can only split the sparse grid in the middle. So this algorithm can only create $2 \cdot D + 1$ smaller grids. Also every grid has a different size, so we get an imbalanced work distribution. The maximum speedup is therefore restricted by the biggest partial sparse grid.

For the hierarchization of a non-zero bounded sparse grid we also have to hierarchize the grids, which represent the border.

// Listing 2.16
public def hierarchize() {
    val index: Array[int] = new Array[int](maxDimension);
    val level: Array[int] = new Array[int](maxDimension);
    for (var d : int = 0; d < maxDimension; d++) {
        for (var i : int = values.size - 1; i >= 0; i --) {
            idx2gp(level, index, i);
            val lParent: Double = parent(level, index, d, -1);
            val rParent: Double = parent(level, index, d, 1);
            values(i) -= (lParent + rParent) * 0.5;
        }
        for (var i : int = 0; i < maxDimension; i++) {
            if (i < d) {
                boundary(i, 0).hierarchize(d - 1);
                boundary(i, 1).hierarchize(d - 1);
            } else if (i > d) {
                boundary(i, 0).hierarchize(d);
                boundary(i, 1).hierarchize(d);
            }
        }
    }
}
5. Parallel Algorithms
Part III.

Results and Conclusion
6. Results

For the tests, we are going to compute an expected value by integrating a five dimensional function. Therefore we are concentrating on hierarchizing five dimensional regular sparse grids of different refinement levels. In some cases we are use sparse grids of other dimensions for comparison. For the speedup and execution time measurements we use the function $f(x) = 16 \cdot \prod_{d=1}^{D} (x_d \cdot (1 - x_d))$. We are going to integrate this function on the domain $[0, 1]^D$. This function is zero at the boundaries and can be used for different dimensions. Because of the predefined grid size at the start of the execution we can use static work distribution.

6.1. Test Environment

As a testing environment the following hardware was used:

- Dual socket Intel SandyBridge-EP Xeon E5-2670, 128 GB RAM and QDR infiniband, 28 Nodes (snb)
- Quad socket AMD Bulldozer Opteron 6274, 256 GB RAM and QDR infiniband, 19 Nodes (bdz)

And the following software was used:

- X10, Version 2.3.1 with C++ backend
- GNU C++ Compiler, Version 4.6
- Java 1.6 (Needed for the X10 Compiler)
- R Version 2.13.1

For the compilation of the X10 code the flags -O and -NO_CHECKS was used. For the C++ implementation the optimazion flag -O3 was used. In order to set up the X10 runtime optimal, we can use the environment variables X10_NPLACES and X10_NTHREADS [22]. By setting the value of X10_NPLACES the runtime starts X10_NPLACES Places at the beginning of the execution. Since a Place represents a virtual shared memory system each Place does not have direct access to the memory of another Place, even if they are running on a real shared memory system. So the best choice of defining X10_NPLACES could be the number of actual shared memory systems available on the machine. By setting the variable X10_NTHREADS we can decide how many worker threads shall run per Place. The runtime can dynamically change the number of worker threads, but will never reduce it to a number smaller than X10_NTHREADS [12]. X10 uses a work stealing scheduler. Therefore idle
6. Results

worker threads are in busy waiting. More worker threads than physical or logical cores can create an overhead. Some additional idle worker thread is always seeking for a new task so it will waste CPU time. On the other side, if there are two tasks for two threads on the same core, they can regularly invalidate each other’s caches. The best choice to define X10_NTHREADS might be the number of available cores for a place.

6.2. Sequential Algorithm

At first we compare the execution time of the X10 and the native C++ implementation. Since the iterative algorithm is supposed to achieve better results than the recursive one\cite{16}, we only analyze the iterative algorithm. For this, we hierarchize sparse grids with increasing grid points. Figure 6.2 shows the absolute execution time for hierarchizing sparse grids of different level and dimensions. For the left graph, only sparse grids with the same refinement level were used. On the right graph, we compare five dimensional sparse grids, with different levels. We can see that the execution of the native C++ code is still faster than X10. But the execution of the X10 code is only slower of a constant factor for the tested sparse grids. This factor is about 2.

![Figure 6.1: This graph shows the absolute execution time for the sequential algorithm of different sparse grids.](image)

6.3. Algorithm for Shared Memory

Now we have a look on the speedup measurements for the shared memory algorithm. In Figure 6.2 we can see that the speedup is almost linear to the number of cores. In comparison to C++ & OpenMP implementation we achieve almost the same results on the AMD Opteron Processors. On the Intel Xeon the C++ & OpenMP implementation achieve better results than the X10 implementation. We can also see that we can also gain an additional speedup on the Intel processors by using Hyperthreads. For this algorithm we achieve for different sparse grids almost the same maximum speedup.
6.4. Algorithm for Distributed Memory

The first algorithm for distributed memory systems, where we distribute the whole grid on all places, achieves similar results when running on a shared memory system. We only create a small overhead by checking the place for a possible remote access. Since we only use one place on a shared memory system this will always be false and we are doing the same as in the shared memory algorithm. By using two or more nodes we are starting to read from a remote memory. So we slow down the execution time caused by serialization of the grid point and latency of the network. Since we distribute all grid points to the different Places, every Place contains about \( \frac{N}{X_{10 \cdot NPLACES}} \) grids points. So we have to read approximately \( N - \frac{N}{X_{10 \cdot NPLACES}} \) per remote access. The number of remote accesses is increasing, but the difference between \( X_{10 \cdot NPLACES} \) to \( X_{10 \cdot NPLACES} + 1 \) is decreasing for increasing Places. So at some point, we can assume that the distribution of additional Places is more significant than the overhead.

We can see in Figure 6.3 that the serialization of the grid points and latency of the net-
6. Results

work slows down our execution for more than two Places. Since the additional overhead decreases for a large number of Places, the execution gets faster. At some point we also reach other boundaries like the bandwidth of the network. This can be seen at the Intel Xeon graph, where the performance on more than 16 Places is decreasing again. Because of the many access we can see that we are about 50 - 200 times slower than the shared memory execution. We can also see, that larger grids also scales worse.

6.5. Algorithm for Distributed Memory 2

The second algorithm for distributed memory stores the whole sparse grid on every Place and only broadcast the results as a big block in each step. This algorithm scales also as good as the algorithm for shared memory, when it is running on a shared memory system. This algorithm requires a lot more memory. We have to store the entire sparse grid on all places. So this algorithm can only be used if the sparse grid is small enough, to be computed on a single shared memory system. The overhead in this algorithm is created by the broadcast of all computations. Our limiting factor is the bandwidth of the communication network. We have to send our computations to all other places, so the overhead increases by the square of the number of places. Because the overhead increases faster than the distribution of work decreases, we obtain at some point a maximum speedup.

Figure 6.4.: This graph shows the relative speedup by using more than one Place. The algorithm for shared memory was used for comparison.

Figure 6.4 shows the relative speedup compared to the shared memory execution. We can see that the speedup rises to some maximum. After that, the overhead slows down the execution. The communication between the different Places is cheaper than the computation for the hierarchization. We can also see that we can gain a better performance for larger sparse grids. For the higher levels of a sparse grid we can compute more points per step and therefore send larger blocks which create a smaller overhead.
6.6. Algorithm for Distributed Memory 3

In order to distribute our sparse grid for the third algorithm, we have to split it into smaller sparse grids. Every splitting results in some overhead, because we have to compute the grid points at the middle twice. Therefore we should try to split the sparse grid as seldom as possible. On the other hand, we should not use more places that sparse grids available. And also the absolute execution time is determined by the biggest sparse grid.

Figure 6.5.: This graph shows the relative speedup by splitting the sparse grid into smaller sparse grids. The shared memory algorithm was used for comparison.

In Figure 6.5 we can see that we hit the maximum speedup for low dimensional sparse grids very early. In contrast, by splitting a high dimensional sparse grid into many pieces, we can achieve better results. This algorithm is the only one that also achieves worse results than the shared memory algorithm, when it is running on a single place.

6.7. Comparison of the Algorithms

In this section we will have a short comparison of our algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Easy to implement, best performance (execution time per core and grid point)</td>
<td>Very high execution time</td>
</tr>
<tr>
<td>Algorithm for Shared Memory</td>
<td>Good scalability</td>
<td>Can only be used on shared memory machines</td>
</tr>
<tr>
<td>Distributed Sparse Grid</td>
<td>Can compute very large sparse grids</td>
<td>Execution time can be worse than sequential execution</td>
</tr>
<tr>
<td>Broadcast Computations</td>
<td>Best results on distributed Memory machines</td>
<td>Big memory overhead</td>
</tr>
<tr>
<td>Grid-Splitting</td>
<td>Distributes the sparse grid without dependencies to each other</td>
<td>Some grid points have to be computed twice. Works best on high dimensional sparse grids</td>
</tr>
</tbody>
</table>
6. Results

6.8. Application

For our application we compute an expected value using canonical vine copulas\cite{5}. The R script uses the package CDVine\cite{18}, available on the Comprehensive R Archive Network (CRAN). We get the raw data for our sparse grid by an interface to the R Runtime. The input is a five and the output is a 100 dimensional vector. So we have to compute 100 times a five dimensional sparse grid. Figure 6.6 shows some of the results of integrating different components on different refinement levels.

Figure 6.6.: This graphs show the result of integrating some of the components of our result vector.

On Figure 6.7 we can see that the execution time for hierarchization is much smaller than the creation of the input data for the sparse grid algorithm.

Figure 6.7.: This graph shows the absolute time for creating the input data and hierarchize the sparse grid.
7. Conclusion

We have evaluated different sparse grid algorithms for the programming language X10. The algorithms are differentiated by the memory architecture where they are running on. We have seen that the sparse grid algorithm for shared memory systems scale quite well. The X10 implementation achieves almost the same results as the C++ implementation does.

For the distributed memory execution we tried three different approaches. All of them created some additional overhead compared to the shared memory algorithm. The second algorithm where every place contains the whole sparse grid achieved the best results on distributed memory systems. This can only be applied, if the whole sparse grid can be stored on the local memory of every machine. If the memory is not big enough we need to use one of the other algorithms. Because of the limited capabilities of the third algorithm, where we split the sparse grid into smaller sparse grids, we can use this algorithm only for smaller distributed memory machines. The power of this algorithm can also be seen on high dimensional problems. In section 6.6 we have seen that the access to a DistArray by using the at keyword can be very inefficient. However performance tuning is part of the ongoing work [23]. The first algorithm should only be used if we cannot use one of the other two. This could be a very large grid on many machines. This algorithm enables us to compute very large sparse grids, but it needs much more time.

We have also seen that for some applications the execution time of the hierarchization is significant smaller than the creation of the data used for the sparse grid. To create more performant assemblies we have to improve our code by using the at accesses more efficiently. Another possible solution is given that X10 is improving the implementation of the at accesses.
7. Conclusion
Appendix
A. X10 Implementation for the second Distributed Memory Sparse Grid Algorithm

```java
import x10.util.Timer;
public class Dist2{
    static val maxLevel:int = 10;
    static val dimension:int = 10;

    /**
     * Zero−Bounded d−dimensional parabolic test function.
     * The function maps a point from the domain [0,1]^d
     * to a scalar in R, where every point at the boundary has
     * the value 0. It works on any dimension given by
     * the size of the pos array.
     * Below some integrals expected for dimension d:
     * Integral d=2 -> 0.44444
     * Integral d=3 -> 0.07406
     * Integral d=4 -> 0.01234
     *
     * @param pos The position where to evaluate the function
     * @return The result for the given position
     */
    public static def evalFunction(pos:Array[Double]):Double {
        var result:Double = 16;
        for (var i:Int=0; i < pos.size; i++){
            result *= pos(i);
            result *= (1 − pos(i));
        }
        return result;
    }
}
```

/**
 * Test function evaluated by the level and index vector.
 * The position vector gets generated by the level and index
 * vector. It follows the rule pow=0.5^level*index on each
 * component.
 */
public static def evalFunction(level: Array[int],
   index: Array[int]): Double {
    var result: Double = 16;
    for (var i: Int = 0; i < level.size; i++) {
        var fac: Double = Math.pow(0.5, level(i)) * index(i);
        result *= fac;
        result *= (1 - fac);
    }
    return result;
}

private static def idx2gp(level: Array[int], index: Array[int],
   idx: int) {
    var sumLevel: int = maxLevel - 1;
    var temp: int = idx;
    for (var t: int = 0; t < maxLevel; t++) {
        if (index3Lookup(t) <= idx)
            sumLevel = t;
    }
    temp -= index3Lookup(sumLevel);
    var temp2: int = temp >> sumLevel;
    temp -= temp2 << sumLevel;
    for (var t: int = dimension - 1; t > 0; t--) {
        temp2 -= binomial(t + sumLevel, t);
        for (var i: int = sumLevel; i >= 0; i--) {
            if (temp2 + binomial(t + sumLevel - i, t) >= 0) {
                level(t) = i + 1;
                sumLevel -= i;
                temp2 += binomial(t + sumLevel, t);
                break;
            }
        }
    }
}

* @see evalFunction(Array[Double])
* @param level, index The position where to evaluate the function
* @return The result for the given position
*/
level(0) = sumLevel + 1;

for (var t:int=dimension−1; t>=0; t--){
    index(t) = ((temp & ((1 << (level(t)) − 1) − 1)) << 1) + 1;
    temp >>= (level(t) − 1);
}

/**
 * Implements the gp2idx function.
 * gp2idx is a bijektiv function that maps perfectly every grid
 * point of a fixed depth sparse grid to a 1d array.
 * For the algorithm and further information check the following
 * link:
 * http://dl.acm.org/citation.cfm?doid=1941553.1941559
 *
 * @param [in] level, index The grid point which shall be mapped
 * @return The index where the grid point is stored
 */
private static def gp2idx(level:Array[int], index:Array[int]):int {
    var index1:int = 0;
    for (var t:int = 0; t < dimension; t++){
        index1 <<= (level(t) − 1);
        index1 += (index(t) − 1) >> 1;
    }
    var sum:int = (level(0) − 1);
    var index2:int = 0;
    for (var t:int = 1; t < dimension; t++){
        index2 -= binomial(t + sum, t);
        sum += (level(t) − 1);
        index2 += binomial(t + sum, t);
    }
    index2 <<= sum;
    var index3:int = index3Lookup(sum);
    return index1 + index2 + index3;
}

/**
 * Returns the index of a parent node of the given point.
 *
 * @param [in] level, index The grid point whose parent is expected
 * @param dimension The dimension where to search the parent
 * @param direction The direction for the search. −1 is for
 * the left Parent and +1 is for the right
private static def parent(level: Array[int], index: Array[int],
dimension: int, direction: int): double {
val tempLevel: int = level(dimension);
val tempIndex: int = index(dimension);
index(dimension) += direction;
if(index(dimension) == 0 ||
index(dimension) == (1 << level(dimension))) {
index(dimension) -= direction;
return 0;
}
while ((index(dimension) & 1) == 0) {
index(dimension)>>=1;
level(dimension)=1;
}
val levelSum = level.reduce((a: Int, b: Int)=>(a+b),-level.size);
val idx = gp2idx(level, index) - index3Lookup(levelSum);
level(dimension) = tempLevel;
index(dimension) = tempIndex;
return values(levelSum)(idx);
}

static val binomial = new Array[int](
    Region.makeLowerTriangular(maxLevel + dimension));
static val index3Lookup = new Array[int](maxLevel+1);
static val powerOf05Lookup = new Array[Double](maxLevel +
dimension);
static val values = new Array[Array[Double]](maxLevel);

public static def main(args: Array[String]) {
val numThreads = Int.parse(args(0));
Console.OUT.println("LevelCopyAdvance");
Console.OUT.println("Places = " + Place.MAX_PLACES);
Console.OUT.println("Threads = " + numThreads);
Console.OUT.println("Start d="+dimension+", l="+maxLevel);
finish for(val p in Place.places()) async at (p) {

    // creates the binomial coefficient lookup table
    binomial(0, 0) = 1;
    for(var n:int=1; n<maxLevel+dimension; n++) {
        binomial(n,0) = 1;
        binomial(n,n) = 1;
        for(var k:int=1; k<n; k++) {
            binomial(n, k) = binomial(n-1, k-1) + binomial(n-1, k);
        }
    }
}
// creates the index3 lookup table
var count : int = 0;
for (var i : int = 0; i < maxLevel; i ++){
    index3Lookup[i] = count;
    var add : int = binomial(dimension - 1 + i, dimension - 1) << i;
    count += add;
    values[i] = new Array[Double](add);
}
index3Lookup[maxLevel] = count;

// creates lookup tables for power of 2 and power of 0.5
var pow : int = 1;
for (var i : int = 0; i < maxLevel + dimension; i ++){
    powerOf05Lookup[i] = 1. / pow;
    pow <<= 1;
}
val numValues = index3Lookup[maxLevel];
Console. OUT.println("numValues = "+numValues*0.000001+" mio");
var time1 : long = 0;
var time2 : long = 0;

// fill array with values
val time1 = Timer . milliTime ();
finish for (val l in values){
    val index : Array[int] = new Array[int](dimension);
    val level : Array[int] = new Array[int](dimension);
    for (val p in values(l)){
        val idx : int = p(0) + index3Lookup(l(0));
        idx2gp(level, index, idx);
        values(l)(p) = evalFunction(level, index);
    }
    for (val p in Place . places()){  
        if (p!= here)
            Array . asyncCopy (values(l), 
            at (p) (new RemoteArray(values(l))));
    }
}
val time2 = Timer . milliTime ();
Console . OUT.println(" fill ");
// start the hierarchization

time1 = Timer.milliTime();
for (var d: int = 0; d < dimension; d++) {
    val dim = d;
    for (var i: int = maxLevel - 1; i > 0; i --) {
        val l = i;
        val offset = index3Lookup(l);
        finish for (val p in Place.places()) async at(p) {
            val size = (values(l).size / Place.MAX_PLACES) +
                        ((values(l).size % Place.MAX_PLACES == 0) ? 0 : 1);
            val rangeStart = Math.min(values(l).size, p.id * size);
            val rangeStop = Math.min(values(l).size, ((p.id + 1) * size));
            val rSize = rangeStop - rangeStart;
            val sizePerThread = (rSize / numThreads) +
                               ((rSize % numThreads == 0) ? 0 : 1);
            finish for (var t: int = 0; t < numThreads; t++) {
                val t2 = t;
                async {
                    val index: Array[int] = new Array[int](dimension);
                    val level: Array[int] = new Array[int](dimension);
                    val threadStart = rangeStart +
                                      Math.min(rSize, t2 * sizePerThread);
                    val threadStop = Math.min(rangeStop, threadStart + (t2 + 1) * sizePerThread);
                    for (var j: int = threadStart; j < threadStop; j++) {
                        idx2gp(level, index, j + offset);
                        val lParent: double = parent(level, index, dim, -1);
                        val rParent: double = parent(level, index, dim, 1);
                        values(l)(j) -= (lParent + rParent) * 0.5;
                    }
                }
            }
        }
    }
}
for (val pl in Place.places()) {
    if (pl != here)
        Array.asyncCopy(values(l), rangeStart, 
                        at(pl) (new RemoteArray(values(l))),
                        rangeStart, rangeStop - rangeStart);
}

time2 = Timer.milliTime();
Console.OUT.println("time(\"+d\") +((time2-time1)*0.001));
}
time2 = Timer.milliTime();
Console.OUT.println("hierarchize");
Console.OUT.println("time: "+((time2-time1)*0.001));

// Integrationstest
time1 = Timer.milliTime();
var result:double = values(0)(0) * powerOf05Lookup(dimension);
for(var level:int= 1; level < maxLevel; level++){
    var sum:double = 0;
    for(var i:int= index3Lookup(level);
        i<index3Lookup(level+1); i++)
        sum += values(level)(i-index3Lookup(level)) ;
    result += sum * powerOf05Lookup(dimension+level);
}
time2 = Timer.milliTime();
Console.OUT.println("integrate time: " +((time2-time1)*0.001));
Console.OUT.println("integral: \"+ result);
Bibliography


