IMPLEMENTATION OF DATA PARALLEL PRIMITIVES ON MIMD SHARED MEMORY SYSTEMS

Christian Mortensen
cmn16002@student.mdh.se

Examiner: Björn Lisper
Mälardalen University, Västerås, Sweden

Supervisor: Daniel Hedin
Mälardalen University, Västerås, Sweden

June 3, 2019
Abstract

This thesis presents an implementation of a multi-threaded C library for performing data parallel computations on MIMD shared memory systems, with support for user defined operators and one-dimensional sparse arrays. Multi-threaded parallel execution was achieved by the use of the POSIX threads, and the library exposes several functions for performing data parallel computations directly on arrays. The implemented functions were based on a set of primitives that many data parallel programming languages have in common. The individual scalability of the primitives varied greatly, with most of them only gaining a significant speedup when executed on two cores followed by a significant drop-off in speedup as more cores were added. An exception to this was the reduction primitive however, which managed to achieve near optimal speedup in most tests. The library proved unviable for expressing algorithms requiring more than one or two primitives in sequence due to the overhead that each of them cause.
# Table of Contents

1 Introduction ................................. 1  
   1.1 Problem Formulation ..................... 1  

2 Background ................................ 2  
   2.1 Explicit Parallelism ..................... 2  
   2.2 Compilers ................................ 2  
      2.2.1 Automatic Parallelization .......... 2  
      2.2.2 Parallel Programming Languages ... 3  
   2.3 Data Parallel Primitives ............... 3  
      2.3.1 Communication Primitives .......... 3  
      2.3.2 Element-wise Application of a Higher Order Function .... 4  
      2.3.3 Replication ......................... 4  
      2.3.4 Masking .............................. 5  
      2.3.5 Scan and Reduce .................... 5  

3 Related Work .............................. 6  
   3.1 SkePU and Skeleton Programming ......... 6  
   3.2 TAIL ................................... 6  
   3.3 Data Parallel Languages ............... 6  
      3.3.1 Fortran D and High Performance Fortran ....... 7  
      3.3.2 NESL ................................ 7  
      3.3.3 ZPL and Chapel ...................... 8  
      3.3.4 Futhark ............................. 9  

4 Method ................................... 9  
   4.1 Constraints .............................. 10  

5 Implementing the Library ................. 10  
   5.1 Array Representation .................... 10  
   5.2 Organizing the Threads .................. 11  
   5.3 Distributing the Work ................. 11  
   5.4 Implementing the Primitives .......... 12  
      5.4.1 Get ................................. 12  
      5.4.2 Send ................................. 13  
      5.4.3 Assign ............................... 13  
      5.4.4 Replicate ............................ 13  
      5.4.5 Map ................................. 13  
      5.4.6 Reduce .............................. 14  
      5.4.7 Scan ................................. 14  
      5.4.8 Other Primitives .................... 14  

6 Results .................................. 14  

7 Discussion ............................... 19  

8 Conclusions .............................. 21  
   8.1 Future Work ............................. 21  

References ................................ 23  

Appendix A Parallel Absolute Values ...... 24  
Appendix B Parallel Dot Product .......... 25  
Appendix C Parallel FIR Filter .......... 26
1 Introduction

The transition from a strictly sequential programming model to a parallel one is a process that many find difficult, and because of this some software solution which abstracts the communication between the underlying processing elements of the system is necessary. One such solution which can be used for processing large amounts of data is to use *data parallel programming languages*, which simplifies the process of writing safe and efficient parallel programs by exposing a set of implicitly data parallel constructs (such as loops) and functions. The task of efficiently distributing workloads over all available processors and managing the communication between them is left to the compiler, which apart from increasing the programmers productivity and preventing the large amount of non-deterministic bugs that may occur when using explicitly parallel programming models, also gives the possibility of generating code for several different types of hardware and architecture targets.

Data parallel languages come in many different flavours. Some take an imperative approach with implicitly parallel *forall* loops where each iteration becomes a separate task in itself (such as High Performance Fortran [1] and Chapel [2]), and others follow a more functional route where arrays are directly passed to data parallel *functions* (NESL [3], Futhark [4]). Both of these approaches contains the same underlying concepts and can be reduced to a set of common *primitives* which operate directly on sets of data, including reductions, element-wise applied operations, communication operators for transferring data between data sets, filtering of data based on some boolean mask or predicate, and replicative operations for transforming smaller data sets into larger ones [5].

This thesis describes the implementation of these primitives as a C library targeting shared memory systems, using POSIX threads (pthreads) to achieve multi-threaded execution. Section 2 provides some background to task parallel and data parallel concepts, along with a closer look at the data parallel primitives in section 2.3. Related works are described in section 3, along with a study on earlier implementations of data parallel languages. The methods used for implementing the library and for evaluating it can be seen in section 4, with the actual implementation process is described in section 5. Section 6 contain the results from evaluating the primitives, and these results are discussed in section 7, with conclusions and future work in section 8.

1.1 Problem Formulation

Considering the set of operators that are frequently found in data parallel languages, one can argue that the underlying implementations of these could be reduced to a set of generalized data parallel *primitives*. By studying previous data parallel languages one will find that the execution of the operators exposed to the user can be broken down into several high-level steps. Consider a data parallel vector-scalar multiplication, here the scalar will first need to be copied and distributed to a new vector of the same size as the input vector, which then is followed by an element-wise multiplication across the entire set of elements contained in both vectors.

This thesis will try to answer the following questions: which set of primitives is sufficient to implement a generalized data parallel language, how can these primitives be implemented, and how do they perform in relation to one another when executed on different hardware? The chosen method involves developing a prototype C library containing a set of data parallel functions (or primitives) for multi-core systems by the use of POSIX threads, and as such the primary focus of this thesis are MIMD (Multiple Instruction, Multiple Data) systems with shared memory. This may provide some interesting insights on how the different primitives perform on such systems in terms of both overhead and execution speed, and the implementation will therefore be evaluated with this in mind.

2 Background

This section describes some of the techniques which can be used for parallelizing a program. Section 2.1 provides an overview of explicit parallelism, meaning that execution of a program over multiple processing units is achieved by calling specific communication and synchronization directives. Section 2.2 describe automatic parallelization and data parallel programming languages, where the compiler is put in charge of organizing the communication between the various resources
of the system. Section 2.3 presents some of the data parallel primitives which are relevant for this thesis work.

2.1 Explicit Parallelism

One possible technique for programming parallel machines is to leave the work of modelling the communication between the available physical resources to the user, which is beneficial when full control over the parallel execution of a program is needed. However, this method often require that extensive work be put in ensuring that the processors are properly synchronized to avoid non-deterministic bugs which may be hard to detect and rectify as the code base grows in size. One example of such a bug is a race condition, where several processors manipulate data located at the same memory address at the same time, resulting in a computation which may differ depending on the execution order of said processors [6]. Processor synchronization contribute in large part to the parallel overhead, which must be minimized for the program to achieve the lowest possible execution time. One of the biggest contributing factors to the execution time is usually the time processors spend waiting for others to finish their computations, keeping as many processing units active at all times means that work needs to be distributed effectively among them in order to balance their workloads [6, 115].

There are several APIs and libraries available in order to achieve parallelism by calling explicitly parallel functions, these libraries contain helper functions for communication and synchronization and may operate in a shared or distributed memory. For shared memory POSIX threads (often referred to as pthreads) may be used, which provides the user with various functions for creating and managing threads (several separate control flows within a process) [6, 280]. When using pthreads the responsibility of identifying and securing critical segments in the code (meaning sections of the code which may only be accessed by one thread at a time [6, 287-288]) is put on the programmer. Message passing libraries such as MPI may be used for programming distributed memory systems and supplies various routines for transferring data between different processes [6, 241]. The overhead of parallel programs using message passing is often larger than that of multithreaded programs, but due to the separate address spaces there exists no risk that two different processors write to the same memory location. Race conditions still exist in this model however, and might appear when two or more messages are sent to the same processor.

The main issue that both of these methods have in common is that the programmer is put in charge of ensuring that the parallel executes both correctly and in an optimized manner, which may not be a trivial task. Optimizing a parallel program usually demands that algorithms are rewritten for effective execution over multiple processors, as effective parallel algorithms and their sequential counterparts often look very different from one another. Another important aspect to consider regarding the execution time is that the load between processors must be balanced well to minimize the time processors spend idle. Bugs related to the parallel execution are often non-deterministic and because of this hard to discover, but can still have a major effect on the result of the computation when they do occur.

2.2 Compilers

In order reduce the programmers workload related to the problems discussed in the previous section, the task of assigning work and synchronizing the available resources can be left to the compiler which also provides the possibility of generating code for different types of hardware and architectures.

2.2.1 Automatic Parallelization

Automatic parallelization refers to the process where the compiler transforms sequential code written in an already familiar programming language (such as C) to parallel code without the user having to manage the communication between the different processors [7]. In order for such a compiler to produce an optimal parallel version of a sequential algorithm it needs to effectively analyze the code, and make substantial changes to it based on the information gathered from the analysis. This is required as a sequential algorithm often contain major differences compared to its
parallel counterpart (such as an algorithm for matrix multiplication), and due to this one cannot entirely rely on the compiler to always produce optimal parallel code [7].

There also exists parallelizing compilers that are not fully automatic but still handle the lower levels of communication based on hints in the source code, usually in the form of a set of compiler directives (which is the case for compilers implementing the OpenMP standard [8]), that tells the compiler where and how parallelism can be exploited [9]. Using such a method avoids the complex analysis which is otherwise required, but the more directives that is required to be placed by the user the more it starts to resemble a programming language instead [7, 9].

2.2.2 Parallel Programming Languages

Parallel programming languages brings some of the benefits that come with both automatically parallelizing compilers and explicitly parallel programming interfaces. This section will focus on the data parallel subset of such languages.

Data parallel programming languages exploit (as the name suggests) data parallelism and are influenced by the SIMD (Single Instruction, Multiple Data) programming paradigm where many concurrent executions of the same operation are made on large amounts of data over multiple processing elements [1, 10]. Data parallel languages has existed as early as the 60’s with APL [11] and later found its place as a programming model to take advantage of specific distributed memory systems, such as C*, *Lisp, and CM Fortran which specifically targeted the massively parallel Connection Machine line of supercomputers [1, 12, 13, 14]. More recently the development of semiconductor technology has lead to parallel processors finding their way into general-purpose computers, both in the form of many-core CPU’s and massively parallel processors such as GPU’s, leading to the need for new languages that exploit these types of hardware (Futhark [15], Obsidian [16]). A data parallel execution model makes abstraction of the communication between processors possible, and as such the execution of a data parallel program may be represented as a sequential control flow on the source level [1].

2.3 Data Parallel Primitives

This thesis work will take a closer look at a set of typical operations found in most data parallel languages in the form of primitives, and these will be described in the following subsections. Most of these primitives bear close resemblance to some of the higher order functions commonly found in functional languages, and the descriptions below are based on previous research by Lisper [5] on the relation between data parallel and functional languages. I will provide examples which use data structures found in array based languages, it should therefore be noted that how the functionality of these primitives are expressed may vary between languages (such as languages that use lists or sequences as their primitive data-structures as opposed to arrays).

2.3.1 Communication Primitives

The communication primitive get performs a parallel read of the elements of an array based on some mapping function which maps indices from the input array with indices in the resulting array [5]. As an example, let A be the array to be read from and B be the resulting array, then the mapping function f maps indices from A with their respective indices in B, i.e. $B_i = A_{f(i)}$. This is demonstrated in figure 1.
Figure 1: Demonstration of a parallel read (get) where the mapping function is defined as \( f(i) = (i + 1) \mod 4 \), resulting in a cyclic shift of the entire array to the right.

For transferring data between arrays the \textit{send} primitive may be used. This primitive has the side effect of modifying the data in one of the input arrays as opposed to producing an entirely new array [5]. It also has the possibility of race conditions as it sends data between the positions of the arrays based on some function mapping indices from one array to the other. If two or more indices map to the same index many concurrent sends will be performed to the same position, and thus the result of this operation will depend on the execution order of the processors.

### 2.3.2 Element-wise Application of a Higher Order Function

Usually referred to as a \textit{map}, and bears a lot of resemblance to the higher order function found in most functional languages bearing the same name. This primitive takes \( n \) arrays and a function that takes \( n \) arguments as input, and applies the function at each position such that \( z_i = f(a_i, ..., n_i) \), producing an array of the same size as the range of indices where the operation is defined. As a result of this the primitive can either be seen as a unary or a \( n \)-ary function application depending on the number of input arrays. An example where the sum of the rows of two separate arrays are calculated using a binary summation function is demonstrated in figure 2.

\[
\begin{array}{cccc}
  f & A & B & C \\
  (+) & 1 & 4 & 5 \\
  (+) & 2 & 5 & 7 \\
  (+) & 3 & 6 & 9 \\
\end{array}
\]

Figure 2: An example of an element-wise application of a binary summation function on the two input arrays \( A \) and \( B \). Each element in array \( A \) is added to its respective element in \( B \), producing the array \( C \) where \((C_i = A_i + B_i)\).

### 2.3.3 Replication

The \textit{replication} primitive is used to broadcast a single datum to all positions in an array (see figure 3) [5]. A typical application of this primitive is to fill an array with some scalar value when performing some binary operation where one of the operands is an array and the other is a scalar. This may be used to initialize an array with values when a scalar value is assigned to an array (as an example, consider the syntax \texttt{array x = 0} which will initialize \( x \) as an array only containing 0 elements), but can be used in conjunction with a binary operation which will lead to the operator being applied element-wise after the scalar has been replicated to an array [5]. In languages with support for parallel operations on multi-dimensional arrays it may also be possible to replicate entire arrays and can be used to (for instance) transform a vector into a matrix (by replicating it either row-wise or column-wise), one example of a parallel algorithm that may have use of this is matrix-vector multiplication.
2.3.4 Masking

When constructing parallel algorithms it is often useful to mask out some of the elements of the input array based on some boolean mask, which will lead to these masked elements not being involved in later computations (see figure 4). In other languages this operation might be represented as a higher order function, such as the filter function found in Haskell, which takes a predicate function \( (a \rightarrow \text{Bool}) \) and some list of type \( a \), and returns a new list excluding all elements not fulfilling the given predicate \( (\forall x : P(x)) \)

\[
\begin{array}{cccc}
x_0 & x_1 & x_2 & x_3 \\
1 & 0 & 0 & 1 \\
y_0 & y_3 \\
\end{array}
\]

Figure 4: A boolean mask applied on an array where only the positions of the first and last elements holds true, producing a sparse array where only the first \( (y_0) \) and last \( (y_3) \) elements are defined.

2.3.5 Scan and Reduce

The scan primitive (often called prefix sum) is similar to its counterpart of the same name in functional languages, and combines the elements of an array using some associative binary operator \( (\oplus) \) producing an array containing the intermediate results of applying the operator to all previous elements such that:

\[
y_i = \bigoplus_{j=0}^{i} x_j, \quad i \geq 1
\]

The output will be of the same size as the input and the first element will contain its respective element from the input array \( (B_0 = A_0) \), and each following element \( (B_i) \) will contain the result of applying \( \oplus \) to the element at the same index from the input array \( (A_i) \) and the element at the previous position in the output array \( (B_{i-1}) \) [5, 17].

\[
B_i = \begin{cases} 
A_i & i = 0 \\
A_i \oplus B_{i-1} & i \geq 1 
\end{cases}
\]

Some implementations of scan (such as scanl in Haskell) will use an user specified initial value (usually the identity element of the binary operator), and as such the value of the first index will instead be \( B_0 = x \) where \( x \) is the initial value. Scan can be used to produce the sum of an array, where each position of the output array will contain partial sums up until that point.

The reduction primitive is similar to scan but produces a single scalar value by combining the elements of the input array. The value returned from reducing an array is the same as performing a scan on the same array (using the same operator) and returning the value of the last element in the output array. Reduce can be seen as a binary tree where each leaf is a value from one of the array elements, and each internal node is a binary operator (see figure 5 for an example).
Representations of reduce also exists in functional languages where it is usually called a *fold*.

### 3 Related Work

In this section I will present previous research which is relevant to this thesis work. Section 3.1 presents an implementation of a library for "skeleton programming", exposing a set of functions performing many of the same operations described in section 2.3. Similarly, section 3.2 describes research on an intermediate representation for data parallel languages which also consists of some of these primitives. Both of these section are closely related to the work presented in the rest of the thesis, while section 3.3 contains a study in already developed data parallel programming languages which are not as closely related (but still very relevant).

#### 3.1 SkePU and Skeleton Programming

Skeleton programming is the concept of expressing parallel programs as a sequence of pre-determined higher order functions, so called *skeletons*, that can be translated to efficient parallel code depending on the targeted hardware [18, 19]. As only a certain number of skeletons are available they need to be able to express a wide range of parallel computations for them to be useful. SkePU is a C++ template library that implement this approach, providing a set skeleton functions for performing parallel operations on aggregate data structures such as vectors and matrices. These include *map*, *reduce*, and *scan*, which are similar to the primitives described in section 2.3 and is sufficient to express many data parallel algorithms [19, 20]. In addition to the more general data parallel operations, SkePU also provides specialized skeletons for performing other common instructions such as a combined map and reduce (i.e a MapReduce), and a combination of a *map* and a stencil operation (where each resulting element is a function of several elements from the input vector) [19, 20].

#### 3.2 TAIL

TAIL is a typed intermediate language for data parallel programming, mainly focused on acting as an intermediate representation on the programming language APL [21, 11]. The core language feature many of the data parallel primitives that we have come to expect, such as operators for reducing, mapping (referred to as *each*, which is in-line with its respective APL representation), and scanning [21, 22]. In [22] a translation from TAIL to Accelerate (a Haskell-embedded array language) is demonstrated, showing its viability as an IR for a high level parallel language such as APL.

#### 3.3 Data Parallel Languages

Some not as closely related works are of course data parallel programming languages. There exists a wide range of such languages, targeting different kinds of hardware and expressing data parallel computations in many different ways. Some taking an imperative approach with implicitly parallel looping constructs (Fortran D [23], High Performance Fortran [3]), or by providing a number of data parallel array operators (APL [11], NESL [3], Futhark [4], ZPL [7]), or as a mix of the two (Chapel [2]). A subset of data parallel languages were designed specifically to take advantage of the massively parallel Connection Machine systems, such as *Lisp [13], C* [12], and CM Fortran [14].

Figure 5: The sum of all elements in an array can be computed using the *reduce* primitive, here represented as a binary tree of pairwise additions.
It is also common to find data parallel extensions to existing languages, such as various Haskell
extensions and dialects (Data Field Haskell [24], Data Parallel Haskell [10], Obsidian [16]).

In the following sections a selection of such languages will be examined more closely to try pro-
vide a better view of the state of the art, and give a better understanding of their implementations
and relations to the primitives described in section 2.3.

3.3.1 Fortran D and High Performance Fortran

Fortran D [23, 25] and High Performance Fortran (HPF) [1] are high-level data parallel Fortran
extensions that target MIMD distributed memory machines. Programs written in both of these
languages provide the user with a shared memory view of the program despite the physical memory
being distributed, and parallel programs appear as consisting of only a single thread of control [1].
The core problem which these languages attempted to amend were the difficulties related to se-
lecting the right data distribution when programming distributed memory systems, as once a good
data distribution is found the parallelization of the program is reduced to the task of minimizing
the communication between the various processing units [1]. Arrays to be used in parallel com-
putations are aligned to programmer defined index domains (templates of sorts that are referred
to as decompositions in Fortran D), and these are then used to map array elements to the finite
processing elements of the machine [23]. The actual mapping of arrays to the processors follow one
of three possible pre-defined distribution schemes (block, cyclic, or block-cyclic) that are assigned
to the dimensions of the decomposition [23].

Parallel algorithms in these languages are written with a single thread of control in mind,
but unlike other languages that take a more "functional" approach to data parallelism they also
contain direct references to array elements in the code. A typical data parallel algorithm may
involve looping through an entire decomposition and operating on the individual elements, which
have been spread across the processors according to the selected distribution schemes.

All concurrent computations are handled using the owner-computes strategy where the proces-
sor who "owns" the output of a computation is given the task of actually performing the computa-
tion [1]. The earliest implementations followed this formula rigorously using the left-hand-side
owner-computes rule, which means that the processor owning the data on the "left hand side"
of an assignment statement does the computation. Later implementations (such as that found in
HPF) used a generalized version of this rule however, where any processor which would yield the
highest locality of reference performs the computation [1].

3.3.2 NESL

One of the biggest problems that appears in regards to data parallel programming languages is
how to efficiently perform computations on irregular data structures such as sparse matrices
where some of the elements may be empty (or 0), nested data parallel languages such as NESL allow for
recursive data structures which can be used to represent such structures [3]. The main primitive
data type for parallel computations are sequences and the language contains built-in operators and
functions that may be applied to each of element of a sequence in parallel [3]. The data parallel
operations supported by the languages previously discussed (such as HPF in section 3.3.1) that
can be applied to data structures are unable to filter out the elements that are 0 in (for example)
a sparse matrix, and as such some of the parallel processing power will be wasted. In NESL the
programmer may define the rows of a matrix as a sequence, and each row as a sequence of pairs
containing the column index and the value at that position (empty elements are simply omitted in
the matrix definition) [3]. NESL also allows for calls to data parallel functions to be nested within
each other, and this makes it possible to define recursive functions containing calls to data parallel
functions [3].

The NESL compiler does not directly transform the input code to some other format with direct
calls to message passing routines when communication between processors is necessary (unlike
Fortran D [23]), instead it generates an intermediate representation based on the simple stack-
based language VCODE [26, 3]. All the basic types in VCODE are one-dimensional homogenous
vectors consisting of atomic values such as integers, floats or booleans [26, 27]. Another basic
vector type is the segment descriptor which is used to specify the lengths of different partitions
of another vector, and is useful when operating on nested (multi-dimensional) data-structures as
these have to be flattened when being converted to their VCODE representations [26, 3].

VCODE contains instructions for performing various data parallel operations on vectors [27],
an overview of each of these is given below:

- **Element-wise instructions** - These instructions perform simple operations (including both
  binary and unary operations) on each element in a vector and returns the result [27]. Some
  of the possible operations are element-wise summation of two vectors, or performing an
  element-wise square root operation on a vector [27].

- **Vector instructions** - Included in this set of instructions are such data parallel operations
  as reduce (see section 2.3) and scan [27]. Unlike the element-wise instructions these operate
  on separate vector segments, and therefore require that a segment-descriptor be pushed to
  the stack along with the vector(s) to operate on [27].

The main benefit that comes with generating intermediate code for a language such as VCODE
is that a higher level of machine independence is achieved as the intermediate representation itself
can be compiled to a wide range of parallel machines. VCODE contain only a small number of
simple instructions which makes the task of porting it much easier then porting a higher level
language such as NESL to many different platforms [26].

### 3.3.3 ZPL and Chapel

ZPL is another data parallel language developed for MIMD machines. It shares some of the same
core ideas as High Performance Fortran, such as providing the user with a *global* view of the
program, and the notion of data parallel instructions being executed on user defined index sets (in
ZPL these are referred to as *regions*) [7, 28]. ZPL does not try to extend the syntax of an already
existing language however (unlike HPF and Fortran D which does so to promote code reuse), but
is instead designed from the ground up to provide a set of inherently parallel operations [7, 28].

Parallel arrays are declared by specifying a data type for the elements, and an index set (region) of
some dimensionality and size [7]. Individual elements of the parallel arrays can not be referenced
using scalar indexing (as is the case in HPF), but instead each data parallel operation requires
that the user also specifies a region that tells the compiler at which positions the operations should
be executed [7]. An arbitrary subset of indices in a region can be selected by applying a boolean
mask to each of the positions, and as such all computations involving this new *sparse* region will
only be performed on indices which were not masked out [7].

ZPL provides various scalar operators that are usually found in sequential languages such as
arithmetic, logical, and assignment operators. The main caveat is that these operators use arrays
as operands, and as such are executed in an element-wise fashion [7]. Along with scalar operators,
ZPL also provides some typical imperative control structures (loops and conditionals), as well as
allowing the user to define procedures [7].

To reinforce the idea of operations being applied on large data structures rather than individual
elements a set of *array operators* is made available for transforming the scope of a region, such as
the operator that is used to translate all indices in a region by some vector [7]. The remap operator
is used to (as the name suggests) *remap* the positions of the elements of some array, performing
communication between the processing elements as necessary [29, 7]. Remapping performed on
the right-hand side of an assignment statement is synonymous with the *parallel read* operation
discussed in section 2.3.1, and is done by passing as many mapping arrays as there are dimensions
in the source array. By then combining the elements of these arrays at some position \((i, \ldots, j)\) of
the destination array, the position at which the source array will be read from is produced [29, 7].
Similarly, *parallel send* is performed when a remap is applied to an array on the left-hand side of
an assignment statement [29].

The *flood* operator provides the means of replicating the elements enclosed by a region to some
other region [7]. Various reduction and scan operators also exists which apply a binary associative
operator to each of the elements in some dimension of an array [28, 7]. *Reduce* collapses the
(often) many items in the specified dimension to a single value using the binary operator, while *scan*
instead produces an array of the same dimensionality containing the intermediate results of
each application of the operator in the scanned dimension [28].
Chapel is a language that builds on many of the ideas that made up ZPL but tries to amend many of its drawbacks, such as the lack of support for task level- and general parallelism [2]. The core of the language contains a syntax that in many ways resemble C, with some added quality of life features like type inference [2]. Rather then only allowing for data parallel computations, the language allows for general purpose parallelism through the creation of tasks that simply executes some number of statements asynchronously and then terminates when finished. Tasks may still be synchronized by the use of the sync statement, which typically encloses some number of tasks that has to terminate before proceeding [2].

The data parallel portion of Chapel consists of implicitly parallel forall loops over some range of indices, where a task is created for each iteration [2]. Data parallel operations may either be carried from inside a forall loop by applying scalar operators on array elements selected by their index (array indexing is another major derivation from ZPL), or by promoting a scalar function to instead be applied element-wise to some array (which is done by simply passing an array to a function that expects a scalar value) [2]. Regions from ZPL has been replaced with domains and provide an easy way to name and store index sets to perform data parallel operations upon, these can either be directly passed to a forall statement, or be used to initialize an array [2].

3.3.4 Futhark

One of the more recent directions that research in data parallel programming languages has taken is that of languages specifically targeting other kinds of processing hardware such as GPUs (graphics processing units). A GPU is massively parallel in nature due to their many times higher amount of cores in comparison to a modern CPU, making them excellent tools for fine-grained data parallel processing. Historically these devices were mainly used for real-time graphics processing, but have recently evolved into being a viable choice for general purpose parallel programming with the release of APIs such as CUDA and OpenCL [4, 30].

Futhark is one of the more recent data parallel languages that focus on generating high-performance code for GPUs, and is intended to be used for the more computationally intensive parts of an application that is in large part written in some other general purpose language [4, 15]. Futhark is a purely functional language with a syntax closely related to Haskell, and builds upon the core ideas of flattening nested parallelism from NESL [4, 15]. Several data parallel constructs which are semantically similar to the primitives discussed in section 2.3 are exposed to the user. These include reduce, scan, replicate, map (which is equivalent to the element-wise application primitive described in section 2.3.2), and filter (which produces an array containing the elements of another array where some predicate holds true) [4].

4 Method

To specify the set of primitives that make up a generalized data-parallel language we look at the languages described in section 3.3. We see that most of them contain collective operators (such as scan and reduce), communication operators (that perform actions such as manipulating index domains and sending elements to other arrays), and element-wise instructions (such as the traditional map found in functional languages, or scalar operations applied to each opposite element in two arrays). Those languages that support sparse arrays also have some sort of masking primitive for removing certain elements from a computation.

The implementation supports computations on one dimensional sparse arrays, and the set of primitives that were chosen to be implemented can be seen in table 1. These includes primitives previously described in section 2.3, along with some additional ones such as count which returns the number of elements in a sparse array (excluding those that have been masked). The masking of elements is not implemented as a separate primitive, but is instead a part of every other primitive that may need it.

The viability of using these primitives to express data-parallel programs was evaluated by various experiments carried out on a C library that allows the user to formulate data-parallel programs as a sequence of calls to explicitly parallel functions (representing the primitives). POSIX threads (pthreads) was used to achieve a multi-threaded execution, where threads are created and destroyed automatically, as well as synchronized at various points during the execution of a
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>Takes elements from some array and returns a new one.</td>
</tr>
<tr>
<td>send</td>
<td>Sends elements from one array to another.</td>
</tr>
<tr>
<td>assign</td>
<td>Takes two arrays and assigns the elements from one to the other where a predicate is satisfied, creating a new array.</td>
</tr>
<tr>
<td>replicate</td>
<td>Takes a single value and creates an array only containing copies of it.</td>
</tr>
<tr>
<td>map1</td>
<td>Applies a function to every element in an array.</td>
</tr>
<tr>
<td>map2</td>
<td>Applies a function element-wise to two arrays where they are both defined.</td>
</tr>
<tr>
<td>map3</td>
<td>Applies a function element-wise to three arrays where they are all defined.</td>
</tr>
<tr>
<td>reduce</td>
<td>Takes an array and applies an associative binary function to each of the elements until it is reduced to a single scalar value.</td>
</tr>
<tr>
<td>scan</td>
<td>Takes an array and calculates the intermediate results of applying an associative binary function across all of its elements.</td>
</tr>
<tr>
<td>concatenate</td>
<td>Appends an array to where the bounds of another ends, creating a new array.</td>
</tr>
<tr>
<td>select</td>
<td>Selects a subset of indices from an array, creating a new array.</td>
</tr>
<tr>
<td>count</td>
<td>Counts the number of (defined) elements in an array.</td>
</tr>
</tbody>
</table>

Table 1: Summary of the primitives exposed by the library.

4.1 Constraints

The library does not allow for polymorphism when it comes to the types held by the arrays, or the types that the primitives operate upon. All such type information is specified at compile time, and limits each program to only use one type as their bearer of parallel data. Nested parallelism is also unsupported, meaning that a primitive cannot be nested within other primitives. Optimization techniques such as effective balancing of the load that each thread is assigned was avoided, and as such is not part of the current implementation.

5 Implementing the Library

In this section I will describe the process of implementing the library, as well as some of the problems that I faced during. In this section I will describe indices in terms of global and local space, where indices in the global space are the ones that the user sees, and local space indices are the actual indices of the “physical” arrays (i.e. the allocated memory).

5.1 Array Representation

Parallel arrays are implemented as a struct with information about the lower and upper bounds of the array, and a pointer to the allocated memory containing the actual data. The bounds are
used to determine the global range of the array (from \( m \) to \( n \)), meaning that elements at positions outside this interval may be ignored during parallel computations as they are not defined. The amount of memory that is allocated for the physical array is just enough to fit these bounds, and the actual indices range from 0 to \( n - m + 1 \) (in local space). This representation may be enough for dense arrays, but the masking primitive produces arrays that are sparse, so some way to tell the undefined positions from the ones that are defined is necessary.

One way to implement sparse arrays is to have a field reserved for some sort of bitmask in the array struct, which tells which positions are defined (1) and which are masked (0). A simpler method, that was actually used, is to use an option type (a common concept in functional languages) for each element in the array. This was implemented as a struct containing the value and a boolean value (an unsigned \( \text{char} \)) representing the element being something or nothing.

The resulting representation is an array ranging from \( m..n \), where each element is something or nothing. Each implemented primitive has to convert from global to local indices when reading and writing to such arrays, and also check if each individual element is defined or not.

### 5.2 Organizing the Threads

The creation, execution, and termination of threads are the three basic steps in a multi-threaded program. The naive solution to implement threading for a library of data parallel functions would be to spawn all necessary threads whenever a parallel function is called, letting each thread compute its assigned workload after which all active threads are terminated. The overhead of constantly spawning and terminating threads would be too large however, so a more efficient and elegant solution is necessary. The solution that was chosen was to implement a thread pool, where all the threads that are needed are created ahead of time (typically one for each processor core), and these threads remain in an idle state until they are called for to perform some computation. This calls for some sort of barrier synchronization, as it is not wise to expect that all threads are ready to perform work at the same time after being instantiated. A barrier is also necessary to synchronize the worker threads (along with the main thread) following a data parallel computation to tell when a final result has been produced. The basic parallel execution scheme is demonstrated in figure 6.

Due to pthreads being used there exists a few different ways of implementing the barrier, however the basic idea is always the same. Threads entering the barrier increment a counter representing the number of threads waiting to synchronize. Once the final thread enters the barrier, all other threads are released and the counter is set to 0. One way of implementing this in practice is to use a combination of pthreads mutexes and conditions, essentially relying on the operating system to block and release the synchronizing threads. This does incur some overhead as the OS puts threads to sleep and wakes them up, which may not be optimal when performing many parallel operations in short succession (or for data parallel operations requiring many different synchronizations to complete). Instead I opted to implement a lock-free barrier as described by Berger and Stamatakis in [31], where threads "busy wait" rather then going to sleep when they need to block. This reduces some of the overhead, but the execution time may prove to be unpredictable depending on the underlying system, as well as wasting processing resources on doing nothing when workloads are unbalanced across the processors.

### 5.3 Distributing the Work

The distribution of work across the threads was done naively as an optimized and balanced execution of the workloads was not a goal for this project. The work is block distributed evenly across the threads based on the index range which is to be operated upon. Each thread will then only perform the work on the range of indices that it is assigned. One of the major issues that comes with using an option type for the individual elements denoting them being masked or not (as discussed in section 5.1), is that the indices that the masked elements occupy does not get taken into account when distributing the work which may lead to the load being unbalanced when arrays containing many undefined elements are used as inputs.

Because the computations are data parallel, all threads can be assigned the same instruction in a SIMD-like fashion. All threads operate on the same range of locations in memory, meaning that they take the same arrays as input and writes their individual results to the same shared output
The threads are assigned their instructions with the use of a single global struct containing pointers to the input and output arrays, the distribution scheme, a pointer the task-level routine for the invoked primitive, and the function pointers passed by the user (as well as their arguments if applicable).

When an instruction has been set, the master thread synchronizes with the workers using a barrier to make sure that they’re ready to start processing the work. Once in sync, each thread invokes the task-level routine from the global instruction struct, passing all other arguments to it as well.

5.4 Implementing the Primitives

All primitives are implemented in a similar way with a function (that is exposed by the API to the user) running on the master thread, and some number of task-level functions for performing the actual work. The ”master” function is responsible for distributing the work as described in section 5.3, calling the thread pool to wake the worker threads, collecting the results calculated by the workers (merging it if necessary), and returning it.

Most primitives also take a pointer to a predicate function that is defined by the user, along with a void pointer for possible arguments. This is how masking is implemented in the current state of the library, and is embedded in every primitive in such a way that each primitive masks out every element not satisfying the predicate. The predicate functions always take a global array index as an argument along with pointers to each input array and arguments, which can be used to either mask out specific indices, or mask out elements containing values not fulfilling some condition.

In the following sections I will go over each of the implemented primitives in turn, a brief summary of them can also be seen in table 1.

5.4.1 Get

Get creates a new array based on some user defined function \( f : \text{int} \rightarrow \text{int} \), that maps indices in the output array to indices from which values will be read from the input array. Memory of the same size as the physical size of the input array is allocated for the resulting array, and the work is distributed evenly according to this. Each thread iterates over the indices in their assigned blocks, and sets the output at the position of each iteration to its corresponding value from the
input array (based on \( f \)). In case an element is masked, it is instead given a \textit{nothing} value based on the option type described in section 5.1. The output array is then simply returned to the user after all threads have synchronized.

### 5.4.2 Send

The send primitive is interesting as it is the only one that directly modifies the input that is passed to it. It takes a destination array, a source array, and a function \( f : \text{int} \rightarrow \text{int} \) that maps indices from the source to indices in the destination array. No memory allocation is done as the threads operate directly on the input. The mask will be applied on indices in the source array if specified, and these masked elements will not be sent to the destination array.

No synchronization is done between the threads (apart from the final barrier when all computations are done), meaning that non-deterministic writes to the same indices in the destination array may happen. It is therefore preferable that the user defined mapping function is bijective in its domain to avoid race conditions.

### 5.4.3 Assign

Assign can be seen as a middle-ground between Get and Send, where the elements from one array is moved to their opposite positions in another array. This bears a striking resemblance to a Send where the identity function is used to map indices from one array to another, but a major difference is that it does not modify the destination array but instead returns a fresh array. The primitive is not very useful on its own as it will just return a copy of array from which the values were sent, but together with a predicate it will contain elements from both arrays (where values from the destination array is placed at masked indices).

### 5.4.4 Replicate

Replication is a simple operation that takes a value and a range of indices, and generates an array bounded by the range where every element contains copies of the specified value. An exception is made for masked indices, as these will be undefined (they are given a \textit{nothing} value). The implementation should not be hard to understand, so it will only be explained briefly. The work is distributed evenly across the threads according to the size of the output array, and each of the threads iterate across their assigned indices and either set the value of each element as defined or undefined depending on the mask at that index.

### 5.4.5 Map

The element-wise application primitive is implemented as three separate functions, each taking a different number of arrays as input. In theory one would probably want a single function capable of taking a variable number of arrays, however due to time constraints and the limitations of the C language they are currently defined as \textit{map1}, \textit{map2}, and \textit{map3} (each taking one, two, and three arrays respectively). All of them also require a function pointer to be passed with as many arguments (of the same type as the arrays value type) as the primitive takes arrays, which is to be applied in an element-wise fashion.

One interesting problem that arises when applying this on more than one array is how to handle arrays of different bounds. A strict approach would only allow arrays of the same size to be mapped over, which is the approach taken for the map skeleton in SkePU [19]. However this a strict solution does not make sense when you consider mapping over sparse arrays bounded by some arbitrary range of indices from \( m \) to \( n \), as if the operation is only be defined for the same ranges of indices, it should also only be defined for arrays where each of the corresponding elements in all input arrays are defined (and undefined) at the same positions. Instead I opted for a solution where the resulting array is bounded by the area in which all the ranges of the input arrays intersect, and where each resulting element is undefined if one or more of the input elements is undefined.

The distribution of the work is divided evenly across the range of intersecting elements, outputting their results to a newly allocated array of the same size, and each thread makes sure that all elements at a given index are defined before applying the user specified function to them.
5.4.6 Reduce

For reduce, we want to take an array of values and reduce it to a single value. This calls for a different approach on how to handle the output of the task-level operations, where each thread reads from an arbitrarily sized block and outputs a single element, as opposed to as many elements as is contained in the block. In a way one could see this operation as the reverse of replication, where a single value is transformed into an arbitrarily sized array.

The chosen solution allocates intermediate array of the same size as the number of worker threads, and each worker is assigned a single element to write to. The operator is sequentially applied to the starting value and each item in the threads assigned block, effectively reducing it to \( n \) intermediate reductions where \( n \) is the number of workers. Elements not fulfilling the predicate are masked, and are simply skipped when applying the operator. After synchronizing the threads the intermediate array is reduced to a single scalar on the master thread much in the same way as before, and this scalar is then returned.

5.4.7 Scan

Scan is similar to reduce, but the implementation contains some key differences where the computation is done in two passes. In the first pass the threads perform the same operations as reduce, but the intermediate results are stored in an output array the same size as the input. This creates a set of local scans, one for each thread, where the total result of applying the function to every element is stored in the last index of each block. The second pass merges the values of the local scans with the final results from the blocks preceding them, and encompasses all blocks except the first one.

The values of masked elements are ignored during the computations, and the output elements at these positions are given the same value as the element before it and will as such only produce dense arrays. The mask is only applied during the first pass.

5.4.8 Other Primitives

Some other primitives are implemented which might be useful for certain algorithms. Concat appends one array to the end of another, creating a new array with the same starting bound \((m)\) as the first array, and the ending bound being \( m + \text{length}(A) + \text{length}(B) - 1 \) where A and B are the arrays involved in the operation. This is done by simply iterating over both input arrays and assigning the element at each iteration to the correct position in the output, effectively performing as many iterations as the length of the output array. Another primitive is select which extracts a section from an array based on a range of indices that the user passes to the function. This primitive also takes a predicate for masking certain elements, which means that it can also double as a stand-alone masking function when the same bounds as the input array is passed. The final primitive count returns the number of elements in an array that is defined.

6 Results

The implementation was evaluated with experiments in terms of benchmarks measuring performance when executed on different numbers of cores or with varying sizes of input arrays. The types of hardware that was involved in the experiments includes both older (Intel Core 2 Quad Q9300) and more recent (Intel i5-7400) processor architectures, as well as those with a somewhat larger (AMD Phenom II X6 1075T) and lower (i5-3230M) number of cores. All processors that the primitives were tested on are listed in table 2.
<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Model</th>
<th>Clock rate</th>
<th>Number of cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>Core 2 Quad Q9300</td>
<td>2.50 GHz</td>
<td>4</td>
</tr>
<tr>
<td>Intel</td>
<td>i5-3230M</td>
<td>3.20 GHz</td>
<td>2</td>
</tr>
<tr>
<td>Intel</td>
<td>i5-6200U</td>
<td>2.80 GHz</td>
<td>2</td>
</tr>
<tr>
<td>Intel</td>
<td>i5-7400</td>
<td>3.00 GHz</td>
<td>4</td>
</tr>
<tr>
<td>AMD</td>
<td>Phenom II X6 1075T</td>
<td>3.00 GHz</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2: The processors that the performance benchmarks were carried out on.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Core 2 Q9300</th>
<th>i5-3230M</th>
<th>i5-6200U</th>
<th>i5-7400</th>
<th>Phenom II 1075T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>get</td>
<td>send</td>
<td>concat</td>
<td>select</td>
<td>map1</td>
</tr>
<tr>
<td>1</td>
<td>4.04</td>
<td>4.02</td>
<td>7.93</td>
<td>3.95</td>
<td>3.97</td>
</tr>
<tr>
<td>4</td>
<td>3.72</td>
<td>3.87</td>
<td>7.59</td>
<td>3.72</td>
<td>3.69</td>
</tr>
<tr>
<td>1</td>
<td>1.61</td>
<td>1.55</td>
<td>2.99</td>
<td>1.42</td>
<td>1.74</td>
</tr>
<tr>
<td>2</td>
<td>1.41</td>
<td>1.35</td>
<td>2.29</td>
<td>1.34</td>
<td>1.37</td>
</tr>
<tr>
<td>1</td>
<td>2.15</td>
<td>2.06</td>
<td>3.99</td>
<td>2.00</td>
<td>2.07</td>
</tr>
<tr>
<td>2</td>
<td>2.10</td>
<td>2.05</td>
<td>3.64</td>
<td>2.04</td>
<td>2.07</td>
</tr>
<tr>
<td>1</td>
<td>1.15</td>
<td>1.15</td>
<td>1.86</td>
<td>0.88</td>
<td>1.16</td>
</tr>
<tr>
<td>4</td>
<td>0.80</td>
<td>0.74</td>
<td>1.64</td>
<td>0.67</td>
<td>0.67</td>
</tr>
<tr>
<td>2</td>
<td>1.09</td>
<td>1.87</td>
<td>3.76</td>
<td>1.91</td>
<td>1.87</td>
</tr>
</tbody>
</table>

Table 3: One core and fully parallel execution times (in milliseconds) of the primitives. Execution times of each processor is displayed column-wise, with the single core performance on the left hand side and with all cores on the right hand side of each column.

The execution times of the primitives can be seen in table 3 and seem to follow the same pattern across all the tested hardware with reduce and count being the fastest by a large margin, with scan and concat being the slowest. Both reduce and count perform very well on hardware with many cores as is evident by examining the execution times column of the six core Phenom II 1075T processor, being close to the performance of the four core i5-7400, and being almost twice as fast as the two core i5-3230M, which in comparison performs much better for all other primitives. Scan and concat both perform double the number of operations as most of the other primitives, so their performance being as poor as the results show comes as no surprise. Interestingly, the three element-wise application primitives map1, map2, and map3 don’t seem to vary too much in terms of execution time despite operating on a different number of arrays. Table 4 shows the relative speedup of each primitive using the formula

\[ speedup = \frac{T_{seq}}{T_{par}} \]

where \( T_{seq} \) is the execution time on one core and \( T_{par} \) is the execution time on all available cores.
Table 4: The speedups gained when executing on all available cores based on the data shown in table 3. Theoretical maximum speedups are displayed in parentheses at the column headers, and is equal to number of available cores. The primitive with the highest speedup on each system is marked in bold.

<table>
<thead>
<tr>
<th></th>
<th>Core 2 Q9300 (4)</th>
<th>i5-3230M (2)</th>
<th>i5-6200U (2)</th>
<th>i5-7400 (4)</th>
<th>Phenom II 1075T (6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>1.09</td>
<td>1.14</td>
<td>1.02</td>
<td>1.43</td>
<td>1.56</td>
</tr>
<tr>
<td>send</td>
<td>1.04</td>
<td>1.15</td>
<td>1.00</td>
<td>1.55</td>
<td>1.73</td>
</tr>
<tr>
<td>concat</td>
<td>1.05</td>
<td>1.31</td>
<td>1.10</td>
<td>1.13</td>
<td>1.36</td>
</tr>
<tr>
<td>select</td>
<td>1.06</td>
<td>1.06</td>
<td>0.98</td>
<td>1.31</td>
<td>1.45</td>
</tr>
<tr>
<td>map1</td>
<td>1.08</td>
<td>1.27</td>
<td>1.00</td>
<td>1.73</td>
<td>1.85</td>
</tr>
<tr>
<td>map2</td>
<td>1.09</td>
<td>1.47</td>
<td>1.06</td>
<td>2.12</td>
<td>2.09</td>
</tr>
<tr>
<td>map3</td>
<td>1.32</td>
<td>1.99</td>
<td>1.26</td>
<td>2.57</td>
<td>2.22</td>
</tr>
<tr>
<td>reduce</td>
<td>2.00</td>
<td>2.13</td>
<td>1.95</td>
<td>4.14</td>
<td>6.42</td>
</tr>
<tr>
<td>scan</td>
<td>1.06</td>
<td>1.23</td>
<td>1.08</td>
<td>1.48</td>
<td>1.60</td>
</tr>
<tr>
<td>count</td>
<td>1.62</td>
<td>1.31</td>
<td>1.20</td>
<td>3.07</td>
<td>5.79</td>
</tr>
<tr>
<td>asn</td>
<td>1.06</td>
<td>1.06</td>
<td>0.94</td>
<td>1.22</td>
<td>1.46</td>
</tr>
</tbody>
</table>

Here we see that reduce is by far the most scalable out of all the primitives, always reaching close to the theoretical maximum speedup on all systems, and even achieving a super-linear speedup on some of them. Count comes in as the second best, which is unsurprising considering the similarities between the operations that both of them carry out. Another similarity between them is that they return scalar values. Because of this the intermediate results of applying a binary function to the elements in the input array can be stored locally until they are finally written back to a single position in the output array, which avoids potential cache misses and cases of false sharing. Primitives that instead return or modify some array perform worse, and my theory is that the main contributing factor to this is that they directly read from and modify their input and output arrays.

Figure 7 show the differences in terms of performance of a subset of the primitives when they are executed over a varying number of threads on the processors with four or more cores.
Figure 7: Execution times of three different primitives over an array of \(5 \cdot 10^5\) elements. No masking was done for any of the primitives, and a function adding two elements was provided to the \textit{reduce}, \textit{map2} and \textit{scan} primitives.

Once again we notice a similar pattern across all the different types of hardware, with the exception of the four core Q9300 processor which seem to perform about the same across all setups.

It is also of value to analyze how these primitives perform in terms of a sequential algorithm. The differences between a sequential algorithm calculating the absolute values of the elements of an array and a parallel version of it using only a map can be seen in figure 8. Both the sequential and parallel algorithm can be seen in appendix A.
Figure 8: Differences in execution times with compiler optimizations between a sequential implementation of an algorithm calculating the absolute values of all the elements in an array of length $9.5 \cdot 10^5$, and its parallel counterpart using only a `map1` to perform an element-wise application of a single input abs function.

Notice that the speedups here are higher then the speedups of `map1` that is displayed in table 4, being about 3.60 as opposed to 1.73 for the i5-7400 and 3.05 as opposed to 1.85 for the Phenom II 1075T. This is most likely due to the more costly operation that was carried out on each of the elements (a conditional statement as opposed to a simple negation that was the previous case). Another thing to notice is that the biggest performance gain is achieved when moving from one thread to two threads, providing almost two times speedup in all cases which is close to the optimum. The speedups of the Core 2 Quad Q9300 stagnate after this point which was the case in the previous benchmarks as well (see figure 7), whereas the other systems show an exponential decrease in speedup as more threads are added.

However for more complex algorithms such as the finite impulse response (FIR) filtering algorithm demonstrated in appendix C, we notice a dramatic difference for the worse in performance between a purely sequential implementation and its parallel counterpart using the implemented primitives, see table 5.
Table 5: Differences in execution time in milliseconds (with and without compiler optimizations) between a purely sequential FIR filter and a version using the implemented primitives on different numbers of threads, with an input array containing $10^6$ elements and 3 weights. A six core AMD Phenom II 1075T was used for this experiment.

The biggest gains in performance comes when transitioning from one thread to two threads, and then drops off steadily as more and more threads are introduced. This is in line with the results gathered from the previous experiments (see figure 8). The execution of the parallel version with optimizations is nearly 20 times slower on one thread as that of the sequential version due to the excessive overhead involved when each primitive needs to execute in sequence, where all active processors are forced to synchronize in-between, and the memory of intermediate arrays storing data necessary to execute the next primitive in the sequence needing to be freed after each iteration of the sequential loop (see appendix C). When executing on the maximum number of cores, the parallel version is still close to 9 times slower then the sequential algorithm with compiler optimizations, which may be an effect of the poor scaling of the array-to-array primitives together with such optimization techniques as vectorization. This clearly demonstrates the downsides of the current implementation.

Unfortunately, this seems to also be the case for a not as complex algorithm as calculating the dot product of two vectors (arrays), which benefits greatly from compiler optimizations. The code used for this test can be seen in appendix B, and the results are displayed in table 6.

Table 6: Differences in execution times in milliseconds (with and without compiler optimizations) between a sequential and parallel dot product algorithm. The input arrays both contain $10^6$ elements each. The test was carried out on a six core AMD Phenom II 1075T.

7 Discussion

The results in section 6 paint a bleak picture of the overall scalability of the implementation, where only the operations that output scalar values (reduce and count) scale well over a large number of cores. Primitives that output or modify arrays perform poorly, but the data displayed in figure 8 show that the primitives that apply some user defined function element-wise (such as the map primitives) scale better the costlier that that function is to execute. As is shown by the much slower execution of the parallel FIR filtering and dot product algorithms, a sequence of calls to data parallel primitives produce far too much overhead for it be a viable alternative to its sequential counterpart.

Another major limitation of the library is the lack of support for multi-dimensional parallel arrays (or parallel arrays where each element is an array in itself). Many of the algorithms that would benefit the most from being rewritten to a parallel version using this library perform some sort of computation on matrices, one example being an algorithm for multiplying a matrix by a vector. Adding support for multi-dimensional would require a quite extensive rewrite of the entire library, as each primitive will need to correctly identify the dimensionality of the input and perform the correct output. For instance, a call to reduce on a two-dimensional array should preferably collapse one of the dimensions of the array, and return a one-dimensional array as opposed to a scalar value.

When it comes to the semantics of the primitives themselves there are a plethora of potential problems to consider before they would be fit to act as a basis for a complete language. For instance, considering that the parallel array representation can be used in a global context but
only a fixed amount of memory is allocated for the data, I can foresee that issues will arise when 
\texttt{send} maps a position from the source array to a position outside the physical boundaries of the 
destination array. One might decide that such a mapping should be forbidden, and as such \texttt{send} 
is limited to only being able to send data to positions that reside within the pre-defined bounds of 
the destination array. The problem with such a solution is that it causes some ambiguities when 
it comes to the representations of the arrays themselves, as they may be sparse and as such may 
contain positions or values that are undefined, but would still be well defined candidates as the 
targets of a \texttt{send}. A similar situation arises when we consider the semantics of a \texttt{map} with a rank of 
two or higher (i.e. \texttt{map2}, \texttt{map3}) applied to arrays of different bounds, is \texttt{map} applied to two arrays 
with different bounds well defined? Should an array be seen as an infinite sequence of potentially 
deﬁned \texttt{elements}, or should it be seen as a bounded range of potentially deﬁned \texttt{values}?

Whichever the case, I don’t think that the current implementation is satisfactory when it comes 
to the amount overhead that the sparse arrays cause, both in terms of memory and work. A sparse 
array is allocated as a contiguous block of \texttt{maybe’s}, where each \texttt{maybe} is represented as a struct 
containing its value, and a boolean marking it as either a \texttt{something} or as \texttt{nothing} (as C does not 
have an actual boolean type, an unsigned char is used for this purpose instead). Each undefined 
position within the bounds of a sparse array still requires as much memory to be allocated as that 
of a defined element, despite not needing any more memory then the size of a boolean. Minimizing 
the memory footprint of the arrays is of great concern as data parallelism reaps the biggest benefits 
when applied to large amounts of data, and in the worst case each array element will require that 
double the size of the actual value type be allocated, depending on the amount of bytes that is 
padded after the boolean. The current representation of the sparse arrays also leads to issues with 
load balancing as arrays are split in their entirety and distributed evenly to the threads, leading to 
some threads potentially receiving a far greater number of undefined elements then the others. A 
future implementation might avoid many of these problems by getting rid of the option type as a 
signifer for sparse elements, and instead use some other data structure pointing out which indices 
are deﬁned or not (such as another array, or a bitmask).

A lot of operations that would be redundant if nested data parallel primitives were supported 
can be found when looking at the code in appendix C. Inside the loop iterating through the length 
of the array of weights we see four separate calls to different primitives, each primitive outputting 
a freshly allocated array.

\(Y\) is at first initialized to an array of zero’s of the appropriate size using the replication primitive, 
and the results of each iteration will be added to this array until the algorithm is finished. The set 
of instructions performed inside the loop is as follows:

1. Each weight is replicated to a new array \(W\) with the same bounds as the array containing 
   the input signal.
2. The input array \(X\) is shifted \(i\) steps to the right, producing a new array \(A\).
3. An element-wise multiplication is performed on \(W\) and \(A\), once again producing a new array 
   \((B)\).
4. Finally \(B\) is added element-wise to \(Y\), and the result of this is assigned back to \(Y\) (accumu-
   lating the sums).

It is not diﬃcult to see why this algorithm would be costly as for each weight four calls to the 
primitives will be made, resulting in a total of 12 calls for the FIR ﬁlter in appendix C which uses 
an array of three weights. An extension to the kind of functions each primitive can take could 
shave oﬀ some of the overhead. One can imagine that a partially applied function could be used 
to eliminate the need for each weight to be replicated into a new array, but is instead added to the 
shifted input array by the means of a call to the \texttt{map1} primitive also taking a partially applied 
addition operator \(+(W_i)\) as an argument. An introduction of a proper type system would probably 
be the natural next step for this to work, and would solve another major issue with the current 
implementation which is that the primitives only operate on one type of data (which is deﬁned at 
compile time).

Theoretically some of the primitives that is part of the current library could be removed, such 
as \texttt{reduce} which could be implemented using a \texttt{scan} instead. The results show that this would be
a bad idea as reduce performs many times better then scan, and as such keeping these separate will provide better performance for array reductions executed on typical multi-core processors.

8 Conclusions

This thesis has presented an implementation of a multi-threaded C library exposing several data parallel primitives with support for user defined operators and sparse array computations. Overhead due to the spawning and terminating of worker threads was avoided by the use of a thread pool, and because of this only an initial set of threads need to be spawned at the start of a program. The primitives are themselves put in charge of block distributing the work across all active threads depending on their individual rules, after which the threads execute the necessary work using the same range of locations in memory as inputs and output. Synchronization between active threads was achieved using a barrier implemented using busy wait to avoid overhead that a mutex solution would cause, but came at the cost of processor time being wasted.

An option type similar to that commonly found in functional languages were used to implement the sparse arrays as arrays of maybes. Each maybe is a struct containing a boolean field marking the element as either some value or nothing. The scaling of the primitives on many-core systems when executed over large workloads were poor with the exception for those outputting scalar values, mainly reduce that managed to achieve super-linear speedups on a majority of the tested hardware. The results indicate that the method used for all array-to-array primitives, where all worker threads iterate through their input data sets and writes the results back to a shared output array at each iteration, is not ideal. Most primitives experienced the biggest increase in terms of performance when executed on two cores, followed by an exponential dropoff as more cores were added. The library proved to be effective for simple computations where only a small number calls to the primitives were required, but were not viable for more complex algorithms due to the excessive amounts of overhead caused by executing a longer sequence of calls.

8.1 Future Work

For the library to encompass all areas of data parallel languages support for nested data structures and nested primitives needs to be added. This will allow computations on arrays with more then one dimensions, and will also reduce some of the overhead required to express certain algorithms if implemented properly but will require a rewrite of large parts of the current system. For general computations some system which allows for polymorphic arrays needs to be added as well, as the current implementation is limited to one type of array per program. The primitives also lack load balancing optimizations, which causes additional overhead when the primitives are executed on arrays where the set of undefined elements are unbalanced across the threads.

The results of the experiments showed that the array-to-array primitives scaled poorly on many cores in comparison to the array-to-scalar ones, and provided at best a two times speedup when executed on six cores compared to the six times speedup that reduce gave. How to effectively implement these array-to-array primitives is a subject for further research, but optimizations definitely need to be made for them to be viable on many-core processor systems.
References


A Parallel Absolute Values

Calculating the absolute values of every element is one of the simplest algorithms you can make with the primitives provided by the library, consisting of a single call to `map1` with a single-value `abs` function passed to it. Both a parallel and sequential version can be seen in figure 9.

```c
// (double -> double)
double abs(double x)
{
    return x < 0.0 ? -x : x;
}
```

a Abs function shared between both implementations

```c
struct _par_array
array_abs(struct _par_array A)
{
    // Apply _abs element-wise across
    // the entire array
    R = vb_map1(_abs, A, null, null);
    return R;
}
```

b Parallel array abs.

```c
struct _par_array
array_abs(struct _par_array A)
{
    // Create an empty array to store
    // the results in. Make it the same
    // size as the input array.
    struct _par_array R = mk_array(NULL, A.m, A.n);
    // Loop through the input array
    for (int j = 0; j < length(A); j++)
    {
        // Make sure the element is defined
        if (IS_SOME(A.a[j]))
        {
            // Apply _abs to the input
            // element, make it defined,
            // store the result in R.
            R.a[j] = SOME(_abs(VAL(A.a[j])));
        }
        else
        {
            // If the element is not defined
            // the resulting element should
            // be the same (i.e. NONE)
            R.a[j] = A.a[j];
        }
    }
    return R;
}
```

c Sequential array abs.

Figure 9: This figure shows the differences between an algorithm calculating the absolute values of an array using only a `map1`, and a purely sequential version (note that it still uses the parallel array struct to be able to support sparse arrays).
B Parallel Dot Product

The dot product of two vectors $x$ and $y$ is defined as:

$$x \cdot y = \sum_{i=1}^{n} x_i y_i$$

A parallel version using the primitives provided by the library is visible in figure 10.

```c
double mul2(double x, double y) { return x * y; }

double sum2(double x, double y) { return x + y; }

double parallel_dot(struct _par_array X, struct _par_array Y)
{
    // Perform an element-wise multiplication of X and Y.
    struct _par_array P = vb_map2(mul2, X, Y, NULL, NULL);
    // Reduce the result of the P to get the summation of the element-wise products
    double res = vb_reduce(sum2, 0.0, P, NULL, NULL);

    // Deallocate the local array contained within P
    free(P.a);
    return res;
}
```

Figure 10: Data parallel algorithm for calculating the dot product of two vectors.
C Parallel FIR Filter

A sequential finite impulse response (FIR) filter can be defined as follows:

\[ y_j = \sum_{i=1}^{k} w_i x_{i+j-1}, \quad j = 1, ..., n - k + 1 \]

where \( x_i \) is an input signal with \( n \) elements, \( y_j \) is the output signal, and \( w_i \) is an array of \( k \) weights.

A parallel version using the primitives provided by the library can be seen in figure 11.

```c
double mul2(double x, double y) { return x * y; }

double sum2(double x, double y) { return x + y; }

// Shifts an array index by some integer
int shift(int i, void* arg) { int s = *(int*)arg; return s + i; }

struct _par_array
FIR_filter(struct _par_array X)
{
    // The number of weights
    int k = 3;
    // Standard C array containing k weights
    double weights[] = { 0.3, 0.5, 1.0 };

    // Initialize the result as an array of the correct size
    // where each element contains a 0.0
    struct _par_array Y = vb_replicate(0.0, X.m, X.n - k + 1, NULL, NULL);

    for(int i = 0; i < k; i++)
    {
        // Replicate the weight of the current iteration over an array
        // with the same size as the input array
        struct _par_array W_i = vb_replicate(weights[i], X.m, X.n, NULL, NULL);

        // Shift the input array (signal) i positions to the right
        struct _par_array A = vb_get(X, shift, (void**)&i, NULL, NULL);

        // Multiply the current weight with the shifted input signal
        struct _par_array B = vb_map2(mul2, W_i, A, NULL, NULL);

        // Add the weighted input to the final result
        Y = vb_map2(sum2, B, Y, NULL, NULL);
    }
    return Y;
}
```

Figure 11: Implementation of a FIR-filtering algorithm using the parallel functions exposed by the barrier. Note that deallocation of memory for the intermediate arrays has been deliberately hidden from the code snippet, but is of course a major source of overhead as \( W_i, A, B \) (and the previous arrays allocated for \( Y \)) will need to be deallocated after each iteration of the for-loop.