KTH

Royal Institute of Technology
Department of Teleinformatics

Data Parallel Programming: A Survey and a Proposal for a New Model

Per Hammarlund and Björn Lisper

September 17, 1993

TRITA-IT-9308

ISSN 1103-535X

ISRN KTH/IT/R -- 93/8-SE

Data Parallel Programming: A Survey and a Proposal for a New Model

Per Hammarlund† and Björn Lisper

† SANS—Studies of Artificial Neural Systems
NADA—Department or Numerical Analysis and Computing Science
Royal Institute of Technology, S-100 44 Stockholm, SWEDEN

Department of Teleinformatics
Royal Institute of Technology
Electrum/204, S-164 40 Kista, SWEDEN

September 17, 1993

Abstract

We give a brief description of what we consider to be data parallel programming and processing, trying to pinpoint the typical problems and pitfalls that occur. We then proceed with a short annotated history of data parallel programming, and sketch a taxonomy in which data parallel languages can be classified. Finally we present our own model of data parallel programming, which is based on the view of parallel data collections as functions. We believe that this model has a number of distinct advantages, such as being abstract, independent of implicitly assumed machine models, and general.

1 Introduction

The benefits and necessities of parallelism have been known for a long time in computer science. Data parallel languages as such started to be discussed explicitly in early to mid 1980. Actually, a data parallel language has been around since mid 1960, namely APL [Iverson, 1962, McDonnel, 1979]. The new thrust in research in data parallel languages started with the introduction of vector processors, like the CRAY, and large SIMD computers, like the Connection Machine [TMC, 1989]. Often data parallel programming paradigms have been introduced to make the compilation or vectorization process easier. In FORTRAN, for instance, it is usually much easier to vectorize a direct operation on arrays than a complicated loop statement. Some algorithms are also easy to express in languages like these, *e.g.* many algorithms in linear algebra.

The increase in data size has also inspired this research. This is very minutely captured in an article about the old C* language for the Connection Machine [Rose and Steele Jr., 1987]:

> "It is not unusual for an application to involve the handling of 10^9 data values. For such applications there is much more to be gained by exploiting parallelism in the data than parallelism in the code."

This survey consists of a literature search and a compilation of references to "data parallel" programming languages described in the literature. First we give a brief description of what we consider to be data parallel processing. We give some examples to illustrate how one can think about this programming paradigm and also what the usual problems and pitfalls of data parallel languages are. Especially we consider what characterizes the usual semantics of a data parallel programming language, and also the benefits of such a language. We discuss the disadvantages of the usually assumed semantics.

Then we proceed with a short annotated history of data parallel languages; describing where and how the languages were implemented. The thoughts that may have influenced their creation are mentioned and also the impact of the languages on the research in data parallel languages.

Third we have tried to classify the data parallel programming languages and group them according to similarity. Within a group the languages rely on the same "basic thought," but even so the differences are pointed out. We also point out the different strengths and weaknesses, as we perceive them, in the languages.

Fourth we present our own model of data parallel programming. Here we look at data parallel programming from an abstract point of view. In this model we do not consider the hardware the programs will be executed on. Instead we consider parallel data sets as functions and data parallel computation as operations on these functions.

2 Basics of Data Parallel Processing

At least three types of parallelism can be distinguished. *Data parallel processing* intuitively implies working on a collection of data in parallel, at the same time. *Control parallelism* is to exploit opportunities for parallelism in control statements, like running both sides of an `if`-statement in parallel. *Process parallelism* is simply loosely coupled processes that (possibly) operate towards the same goal.

We now give a short introduction to the data parallel model of execution. Then we give some examples and discuss data parallel programming starting from those examples.

2.1 Introduction to Data Parallelism

As mentioned above, data parallelism intuitively implies that we manipulate groups of elements and work on data in parallel. Why would we like to handle data as groups and not as individual elements? Well, very often the same operation is applied on all elements of a group of data. In sequential programming languages we must use a loop to apply the same operation to each element in a group, one at a time. In a data parallel language the operation is directly applied in parallel to all elements of the group.

Before we discuss data parallel programming languages, let us first briefly review the basics of *data parallel processing* and then look at a few examples. These examples are intended to show the principle of data parallel processing.

Basics Data parallel processing is to manipulate data in parallel. A fundamental question is then how the parallel data should be organized. A useful abstraction is to think of data parallel entities as a set of *locations*, where each location holds a "scalar" *value*. We refer to a datum in a particular location as an *element*. In an actual parallel implementation, the locations can be memory addresses in different processing units which then allows for parallel manipulation of the elements. The following discussion is based on this view.

Elementwise Operations One fundamental operation of data parallel processing is elementwise application, *i.e.* each element of the datum is operated on in parallel. Assume that we have parallel data $A = \langle 1, 2, 3, 4 \rangle$ and that we would like to add a constant 1 to each element. This computation could be performed according to the following:

$$A + 1 \Rightarrow \langle 1, 2, 3, 4 \rangle + 1 \Rightarrow \langle 1 + 1, 2 + 1, 3 + 1, 4 + 1 \rangle \Rightarrow \langle 2, 3, 4, 5 \rangle. \tag{1}$$

The value 1 is transformed into a parallel datum—it is broadcast to all the locations. We can also consider elementwise computations of parallel data, *i.e.* the data in the parallel data with the *same* location are combined. If we have another group $B = \langle 5, 6, 7, 8 \rangle$ and want to elementwise add the groups, then the computation could be performed like

$$A + B \Rightarrow \langle 1, 2, 3, 4 \rangle + \langle 5, 6, 7, 8 \rangle \Rightarrow \langle 1 + 5, 2 + 6, 3 + 7, 4 + 8 \rangle \Rightarrow \langle 6, 8, 10, 12 \rangle. \tag{2}$$

Any "scalar" function can be elementwise extended, as "+" above, to an operation on parallel data. It is also possible to consider elementwise operations where different "scalar" operations are applied to elements in different locations. Consider for instance a "parallel operation" $F = \langle +, -, *, / \rangle$. Then a possible computation is

$$AFB \Rightarrow \langle 1, 2, 3, 4 \rangle \langle +, -, *, / \rangle \langle 5, 6, 7, 8 \rangle \Rightarrow \langle 1 + 5, 2 - 6, 3 * 7, 4/8 \rangle \Rightarrow \langle 6, -4, 21, 0.5 \rangle. \tag{3}$$

Data Movement—Communication Moving data between different locations is sometimes necessary—sorting the elements of an ordered parallel datum is an example of an operation where this is needed. If the locations are physically placed in *different* processing units, then the data movement implies *communication.* As this is often the case, data movement operations are usually considered to be communication operations. Say that we have A as above and want to reverse the order of the elements in A. Say further that the locations in A are consecutively numbered $0, 1, 2, 3$. Then we can *read* each value of A into the correct location using the mapping $M = \langle 3, 2, 1, 0 \rangle$, where each element of M is seen as a *source adress* where the corresponding data element is to be fetched. More formally, we can consider a parallel operation "$\mathit{read}(A, M)$" which returns a parallel value. This value has the same set of locations as A and M. For each such location i holds that $\mathit{read}(A, M)(i) = A(M(i))$. (We refer to the value of a parallel datum X in location i as $X(i)$.) Thus,

$$\mathit{read}(A, M) = \langle A(M(0)), A(M(1)), A(M(2)), A(M(3)) \rangle = \langle A(3), A(2), A(1), A(0) \rangle = \langle 4, 3, 2, 1 \rangle$$

as desired.

Instead of fetching the elements of A, we can *send* them to suitable *destination adresses.* This is a destructive operation where the destination elements are concurrently updated. (Because of the destructivity it is important that this is done concurrently, as will be discussed in section 2.2.) Formally , we can consider a send operation $\mathit{send}(M, A, B)$ with the effect that afterwards $B(M(i)) = A(i)$ for all locations i. As above, M is a parallel datum, this time holding destination addresses rather than source addresses. In order to reverse A with a send operation we can actually use the same M as above. Then $\mathit{send}(M, A, B)$ yields $B(M(0)) = B(3) = A(0)$, $B(M(1)) = B(2) = A(1)$, $B(M(2)) = B(1) = A(2)$ and $B(M(3)) = B(0) = A(3)$, *i.e.*

$$B = \langle A(3), A(2), A(1), A(0) \rangle = \langle 4, 3, 2, 1 \rangle$$

as desired.

It is commonplace to put restrictions on M. These can arise from physical restrictions in the communication hardware on the machine where the operation is implemented. If, however, the values of M can be independently chosen for any location, then the operation is sometimes referred to as *general communication.*

In a general send operation several data may be sent to the same destination. This happens whenever $M(i) = M(j)$ for some $i \neq j$. What should be done in this case? One strategy is to save only *one* of the arriving data. Another strategy is to somehow combine the data being sent to the same destination—the data can for instance be summed, or concatenated into a list.

Using a sum-combination in a send it is easy to create a *histogram function*, where the elements of the destination are the *bins* of the histogram. For each location, the address is calculated for the bin where the contribution is to be sent. Then, a "1" is sent from each location to the corresponding address.

Code Example 1 A histogramming function written in New C* (see section 4.8). The code uses a sum-combining send.

```
#include <stdio.h>
/* Declare the sizes of the shapes, ie the number of bins in the */
/* histogram and the number of sending PEs. */
#define NOBINS     16
#define NOPEBINS   4096
#define NOSOURCES  8192
/* declare the shapes */
shape [NOPEBINS]histogram, [NOSOURCES]source;
/* declare the data */
int:histogram bins;
int:source sourcevalue;
/* the histogramming function */
void main( void )
{
  int i;
  with( histogram ) bins = 0;           /* reset the bins for a test run */
  with( source )                        /* Set the source data that we would */
    sourcevalue = set_up_source_data(); /* like to histogram. */
  with( source )                        /* do the histogramming */
    [ value_to_bin_n( sourcevalue ) ]bins += 1;
  with( histogram )                     /* print the bins*/
    for( i = 0; i < NOBINS; i++ )
      printf("%4d ", [i]bins );
  printf( "\n");
}
```

Uniform Communication Uniform communication is a form of restricted communication, where all elements concurrently fetch data from (or send data to) the same direction a constant number of steps away. This can be viewed as a *shift* or *rotate* operation on the parallel data. (A special case of uniform communication is *nearest neighbor* communication, where each element fetch data from its immediate physical neighbor. Nearest neighbor communication is usually the generic communication operation on a mesh connected parallel computer.) Note that uniform communication requires some notion of *relative positioning* of the locations. Usually this implies that the locations are points in some regular grid, possibly with torus connections. In the examples in the previous section the locations are the points in the simple one-dimensional grid $0, 1, 2, 3$. Uniform communication in multi-dimensional grids is also possible and in common use.

Relative addressing also introduces the possibility of referring to locations outside the the allowed range. It must be well defined what happens in such cases. Basically two strategies are used. 1. Fetches that go outside return a *specified* element, *e.g.* zero or something defined by the user. This can be viewed as a "shift with zero fill." 2. The locations are considered to be *torus connected*, *i.e.* the accesses wrap around. In the one-dimensional grid $0, \ldots, N-1$, a "to the left"-access a distance d away from location i then goes to location $i - d \bmod N$. This can be viewed as *rotating* the parallel data.

Reduction We often want to combine the elements of a parallel datum into a scalar, *e.g.* by summing it. This is called *reduction.* For instance, we can sum the elements of A above,

$$\mathit{sum}(A) = 1 + 2 + 3 + 4 = 10,$$

or multiply them:

$$\mathit{prod}(A) = 1 \cdot 2 \cdot 3 \cdot 4 = 24.$$

For a parallel datum of size n it is possible to combine the elements sequentially using $n-1$ binary operations. But for all binary, associative operations this can be done in parallel in time $O(\log n)$ by rewriting the "sum" into a balanced binary computation tree. Therefore, reduction operations formed from binary associative operations can be efficiently evaluated in parallel. For floating point arithmetic, changing the order of evaluation can change the result. This happens since the resolution of floating point arithmetic is limited and therefore sensitive to evaluation order: thus, floating point operations are not exactly associative, even though the corresponding arithmetic operations on real numbers are.

If the binary operation is not commutative, then the *ordering* of the elements in the reduction is important. This requires a total ordering on the *locations* of the parallel datum to be reduced. An example of an associative but not commutative binary operation is the *function composition* operation "$\circ$", defined by $f \circ g(x) = f(g(x))$ for all possible arguments x of the functions f and g. It is easy to prove that $(f \circ g) \circ h = f \circ (g \circ h)$ always. On the other hand, $f \circ g \neq g \circ f$ in general. Consider now the parallel datum $\langle f_0, f_1, f_2, f_3 \rangle$ with locations $0, 1, 2, 3$, where the elements are functions. If the result of a reduction with respect to "$\circ$" is to be well defined, a total ordering of these locations is required. If we choose the usual less-than ordering $0 < 1 < 2 < 3$, then the result of the reduction is defined as $f_0 \circ f_1 \circ f_2 \circ f_3$. Another ordering, say $3 \prec 0 \prec 2 \prec 1$, would yield the reduction $f_3 \circ f_0 \circ f_2 \circ f_1$ which is a different function in general.

Running Sums or Scans A generalization of reduction is the *running sum* or *scan.* The running sum of a parallel datum over ordered locations is another parallel datum, where the elements are the successive partial sums from the lower ordered locations and up. *E.g.* with A as above we obtain

$$\mathit{running\text{-}sum}(A) = \langle 1, 1+2, 1+2+3, 1+2+3+4 \rangle = \langle 1, 3, 6, 10 \rangle.$$

The scan operation is running sum generalized to any binary, associative operation.

Again, the scan of a parallel datum of size n is possible to compute sequentially using $n-1$ operations. For all associative binary operations this can be done in parallel by cyclic reduction methods [Kogge and Stone, 1973]. Cyclic reduction on n elements takes $O(n \log n)$ operations and runs in $O(\log n)$ time steps on n processing units. It is possible to implement this very efficiently on a parallel machine with only uniform communication.

The cyclic reduction is performed by every element first fetching the value of its immediate neighbor on the left side and then computing the operation, then every element fetches the (newly computed) value two elements away. If the binary operation has a unit element (like "0" for addition), then fetches that fall outside the location range can return that element. The elements continue like this for $O(\log n)$ iterations, doubling the distance from where they fetch data in each iteration. The two last iterations fetch data from $n/4$ and $n/2$ steps away. The result computed for each location will be the value of the reduction of the elements in the locations up to and including that location. Thus, the total result is the scan. The reduction of the whole parallel datum is found in the rightmost element. Here is an example of a scan (running sum in this case) for a parallel datum with eight elements:

$$\begin{aligned}
\langle 1,2,3,4,5,6,7,8\rangle &\Rightarrow \langle 1+0,2+1,3+2,4+3,5+4,6+5,7+6,8+7\rangle = \\
\langle 1,3,5,7,9,11,13,15\rangle &\Rightarrow \langle 1+0,3+0,5+1,7+3,9+5,11+7,13+9,15+11\rangle = \\
\langle 1,3,6,10,14,18,22,26\rangle &\Rightarrow \langle 1+0,3+0,6+0,10+0,14+1,18+3,22+6,26+10\rangle = \\
&\qquad\qquad\qquad\qquad \langle 1,3,6,10,15,21,28,36\rangle
\end{aligned}$$

2.2 Discussion and Examples

In the previous section we described some basic aspects of data parallel processing. In this section we discuss a little more in detail what data parallel *programming* is, what pitfalls one might uncover, and what semantical constructs that must be present in data parallel languages.

Data parallel processing can be thought of as adding another dimension to the execution of the program: the spatial domain. In a sequential language we have only the time domain to work in, *i.e.* all the operations we do are performed in sequence, one operation at a time. It is easy to understand what happens at any particular time step. In data parallel processing there is also a spatial domain, which means that several data are operated on at the same time, in different places. In order to exploit this we must have a precise understanding of the semantics of the data parallel language.

The first example shows how a loop in a sequential program can be removed when the same program is expressed in a data parallel program—instead of doing the computation in the time domain, looping over the data, we perform it at once in the spatial domain.

In the example we add a constant value to a parallel datum with locations $1,\ldots,n$. The sequential code uses an array ranging over the same locations to keep the data.

Code Example 2 Sequential code for incrementing the value in each element of an array.

```
FOR I=1..N
  DATA[ I ] += CONSTANT
```

Code Example 3 Parallel code for elementwise addition.

```
FORALL IN DATA
  ELEMENT += CONSTANT
```

Let us now consider what has been done here and what semantics have been introduced in these (syntactically) similar examples, without formally stating exactly what the mean:

Time versus Spatial Domain The actual execution of these two examples is entirely different, although they are syntactically similar. In the sequential case we assume that we are using a single processing unit and that the execution takes n time steps. In the data parallel case we assume that the operation uses n processing units in a single time step. In the sequential case the individual increment operations are distributed over the time domain, whereas the increment operations in the data parallel case are distributed over the spatial domain. We simply swap a time domain control construct for a spatial domain control construct.

The semantics of the (sequential) `FOR`-loop is that something that happens in iteration $i+1$ can use results from operations performed in iteration i. In the `FORALL` statement, all increments take place at once. Within this statement, processor $p+1$ cannot use the result of the increment from processor p. This parallel operation is actually a kind of *concurrent assignment*, where a set of variables are concurrently assigned values. [Dijkstra, 1976, Gries, 1978]

Parallel Data In the sequential case each individual element of the array is identified by an index. The corresponding operations are performed in the strict order by which the indices are generated in the loop. Note that we must use this index to refer to the elements one by one even if the operations do not need to be carried out in the particular loop order.

In the data parallel case, we associate the element in each location with the new symbol `ELEMENT`—all the elements of `DATA` are laid out in the spatial domain. The operations performed on the elements can be thought of as being simultaneously performed in spatially separate processing units, each hosting one element of `DATA`. Each "logical" location (corresponding to index in the array) is then translated into the processor address of the hosting processing unit. Note that for the data parallel operation in Code Example 3 no particular order between the elementwise operations is required. This is reflected in the syntax, where the "location identifier" (corresponding to index identifier for arrays) is omitted at the left-hand side of each "local" assignment. Furthermore, it is assumed that `DATA` is defined for exactly the locations $\{1, \ldots, N\}$. This set of locations is a property of `DATA` and it is implicit that all elements of `DATA` are included in the operation. This explains the somewhat terse syntax "`FORALL IN DATA`."

Operators "`+=`" in the parallel case is the ordinary "scalar" increment operator. This is natural here since we in Code Example 3 specify the parallel datum `DATA` and operate on the *elements* of that parallel datum. In each location the element is a scalar. Nevertheless many data parallel programming languages, like *Lisp—see Section 4.4, has special operators for parallel data, even when these operations are composed from scalar operations, *e.g.* elementwise addition is "`+!!`" in *Lisp. This approximately doubles the amount of operators that the programmer has to know.

But using the same function symbols for operations on different types of data can also lead to confusion. Often the rules for *promotion* in programming languages are very poorly understood or even badly implemented. Promotion is the automatic conversion of arguments when they are not compatible with an operator, *e.g.* when adding a float and an integer the integer value will first be converted into a float. The situation is similar in data parallel languages. In Example 1 the expression $A + 1$ results in the value 1 being added to each element in the parallel datum A. To effectuate this the scalar 1 is first promoted to a parallel datum of ones, defined over the same locations as A. If these locations correspond to different physical processing elements, then this scalar-to-parallel promotion can be done with a broadcast, as is discussed in the next section.

Communication Moving the data in `DATA` demands more code in the sequential case. Since there are no concurrent assignments as in the data parallel case, values that are needed later on may be overwritten if care is not taken to save them explicitly.

Assume that the array `M` describes how data should be moved. Each element, with index `I`, should be moved to `M[I]`. The following is an attempt to a sequential program for this:

Code Example 4 Data movement. (This code contains a serious error.)

```
FOR I=1..N
  DATA[ M[ I ] ] = DATA[ I ]
```

This will fail for some patterns of movement, since the values of the array can be overwritten before they are needed. Let `DATA` equal $\langle 1, 2, 3, \ldots \rangle$ and let `M` equal $\langle 2, 3, 4, \ldots \rangle$. If the code above is executed using these data, the value 1 will be written in each element of `DATA`. This is not the desired data shift. A temporary array must be used to explicitly store intermediate results.

Code Example 5 This is a corrected version of Code Example 4. A temporary has been introduced to avoid overwriting values that may be needed later on.

```
FOR I=1..N
  TEMP-ARRAY[ M[ I ] ] = DATA[ I ]
FOR I=1..N
  DATA[ I ] = TEMP-ARRAY[ I ]
```

In fact there is still an error here. If the pattern of movement in `M` is *not* one-to-one, then there are elements in `TEMP-DATA` and `DATA` that are not written to. These should not be modified. To be perfectly safe we must have a third loop before the two above, where `DATA` is copied to `TEMP-ARRAY`.

Consider now the parallel case. Assume that each element in `DATA` resides in a unique processing unit, *i.e.* that each processing unit is uniquely labeled with an array index `I`. Then we can write the following data parallel code for permuting `DATA`:

Code Example 6 Parallel data movement—a send operation.

```
FORALL IN DATA
  DATA[M[I]] = ELEMENT
```

We now assume that there is a communication primitive that can be used to *concurrently* update the elements of the `DATA`. The interpretation of the parallel code above is that the element of `DATA` in processing unit `M[I]` is assigned the value of `DATA` in processing unit `I` (*i.e.* the local "element" of `DATA`). This is done concurrently for all indices where `DATA` is defined, *i.e.* a concurrent send operation. Note that the elements of `DATA` that are not sent to are left intact. (For the time being we ignore the case when several data values are sent to the same element.) Thus, the concurrent send operation eliminates the need to handle temporary storage explicitly, as in the serial case.

It should be noted that the concurrent send operation above need not be implemented as a parallel operation. It could well be implemented on a serial machine, where the array elements reside in the local memory. It is then up to the compiler to arrange the temporary storage that is needed when rearranging the data. Thus, since uninteresting implementation details are hidden, the data parallel primitives described here provide a higher level of programming than the usual primitives for handling *e.g.* arrays.

Broadcasting The symbol `CONSTANT` in Code Example 3 represents a scalar being added to all elements in `DATA`. In the sequential case its value can be accessed directly. But in the parallel case it must be concurrently accessed by all the parallel, spatially separated, processing units. Thus, the scalar must be *broadcast* to all processing units. Broadcasting requires the processors to be operating in a synchronous fashion or that they are synchronized just prior to the broadcast.

2.3 Elementwise vs. Data Field Oriented Data Parallel Programming

The style of programming displayed in the parallel Code Examples above can be coined "elementwise data parallel programming." It is characterized by a "`FORALL`" construct ranging over all locations where a specified parallel datum is defined. Inside a `FORALL` construct the individual elements of the parallel data involved are accessed. Since they are scalars, the operations involved are ordinary scalar operations. This style of programming can be a little confusing at times, especially when expressing operations of a global nature that are not so related to the local elements. An alternative is what may be called "data field oriented data parallel programming," where operations are applied directly to parallel data without any explicit reference to the individual elements. In this style we could write Code Example 3 as follows:

```
DATA += CONSTANT
```

and Code Example 6 as

```
DATA[M] = DATA
```

This style of programming makes much heavier use of promotion rules. In the first example above, `CONSTANT` is first promoted to a parallel datum defined over the same set of locations as `DATA`. The scalar increment operation "`+=`" is then pointwise promoted to a parallel increment operation, that increments each element in `DATA` with the corresponding element of the right-hand side.

2.4 Selection of Elements

Often we like to apply an operator only on some of the elements of a parallel datum. For instance we might want to add 1 to all even elements and subtract 1 from the rest. We can express this control structure in two different ways: 1. A functional approach—we define a function that can be applied locally. This function tests what to do to the elements and it returns a value for each location. It can look like: "`if(odd(element), element+1, element-1)`." 2. An imperative approach—we introduce a new control structure that evaluates the predicate locally and then in a two-step process first turns off the locations with elements not fulfilling the predicate and applies the first operator at the active locations, leaving the others *unchanged*. In the second step the inactive becomes active and vice versa, and then the second operator in applied. These two approaches are functionally equivalent—it is a matter of programming style.

3 Introduction to Data Parallel Programming Languages

In this section we describe some of the basics of data parallel programming languages. Data parallel languages can (crudely) be put in a spectrum with *explicit* languages on one side and *implicit* on the other.

In explicit data parallel programming languages there is a parallel data type, *e.g.* vector or matrix, with special operators. We call these languages explicit since all operations on parallel data are stated explicitly, *e.g.* matrix multiplication or sum. Languages of this type are usually simple to compile since it is known for each operation to what extent it can performed in parallel. Many algorithms can easily and lucidly be coded in an explicit data parallel language, especially algorithms in linear algebra. A problem that occurs when a language is extended with explicit data parallelism is that the number of operators and intrinsics of the language is increased. This, we believe, is a poor state of affairs.

On the other side of the spectrum are the implicit languages. They have no or weakly defined carriers of parallel data. We call these implicit since there are no special parallel operators: instead, the compiler has to find the possible parallelism, *e.g.* by analysing control structures like loops or if-statements. An implicit language may have control structures with strict semantics that helps the compiler to analyze the code. A traditional sequential language is, as far as data parallel processing goes, of course a totally implicit language. Here it is entirely up to the compiler to find what can be done in parallel. A parallelizing compiler for an implicit language can be useful for extracting available parallelism from already existing code, with little modification to the code. If the implicit language is imperative, however, then the use of destructive assignments will create aliasing problems that can make it hard for the compiler to find the available parallelism.

In an explicit data parallel programming language some new control structures must be added to support data parallel programming. Examples are those for communication, for selection of elements, *etc.* These control structures can sometimes replace the traditional ones, like the `FORALL`-statement replaced the `FOR`-loop in Section 2.2. A programming language may include intrinsic operators for non-atomic data types of the language, *e.g.* Common Lisp has summation defined over sequences. If intrinsics on parallel data are defined over the corresponding scalar data types too and if scalar intrinsics are given an interpretation over the parallel data types, then the language becomes more "orthogonal" and therefore easier to learn and use.

It must be considered how a data parallel programming language keeps track of parallel data and how it lets the programmer manipulate and reference these data. Some data parallel programming languages have special objects for handling and storing the parallel data. Such an object has a "shape", which is a set of locations, *e.g.* a shape can be the index range for a one-dimensional vector or the set of integer triples contained in a three-dimensional cube. For all locations in the object's shape, a number of scalar variables can be declared. This kind of object is much like a container of parallel data. This is a useful way to represent parallel data when data are strongly related. Preferably the variables inside such an object should be lexically separate from variables in other objects.[1]

Other programming languages let variables have a shape as a property. Then the shape acts like a data type. An example is ordinary arrays, whose "shapes" are contiguous intervals of integers. For a single parallel entity, each location of the property-shape has only one datum. This is similar to an array. However, the datum at each location can be a record holding several data values: such a parallel datum is then very similar to an "object" of the type described above. Several parallel data may have the same shape: they can then be combined in elementwise operations. The following examples (in hopefully evident pseudo-syntax) shows

[1]This feature of being able to "hide" certain variables puts interesting demands on the communication primitives: they must be able to "see" the variables they are operating on, even though these are not within the scope of the primitive. Should the communication primitives have "X-ray eyes"? Or should the language allow the "export" of identifiers from a shape?

the different styles of representation:

Code Example 7 "Object oriented" style:

```
parallel object x
  {
    shape 1:1000, 5:500
    contains int hugo, nisse, allan
  }
      .
      .
with x do
  {
      .
      .
    hugo = nisse + allan*3
      .
      .
  }
```

Code Example 8 "Shape-as-property"-oriented style (this example uses promotion of scalar operators and constants to parallel data):

```
define-shape Y = 1:1000, 5:500
parallel hugo, nisse, allan: shape Y of int
      .
      .
hugo = nisse + allan*3
```

Code Example 9 "Shape-as-property"-oriented style with records as location values:

```
define-shape Y = 1:1000, 5:500
parallel x: shape Y of record
  {
    hugo:  int
    nisse: int
    allan: int
  }
      .
      .
x.hugo = x.nisse + x.allan*3
```

4 Survey of Data Parallel Languages

This section contains a survey of data parallel programming languages. The languages are presented in roughly the order they appeared in the literature. A few languages are presented in some detail but much of the discussion will take place in section 5 on classification of data parallel languages. Many of the languages described here were created to allow the programmer to use the architecture of a particular machine. While these languages are quite good at controlling the underlying hardware, they usually have poor theoretical foundations.

For some of the languages we supply code examples. These examples are intended to convey the flavor of the language. The examples all show the creation of three parallel variables and how to perform one pointwise multiplication and one global reduction operation on the parallel data.

4.1 APL

Data parallel programming is in many ways a new field. But a survey of data parallel programming languages has to mention APL—*A Programming Language*. APL is one of the first programming languages to support operations on ordered collections of data, namely arrays. APL started as a notation to describe vector computations on the blackboard. The first implementation similar to APL today was available in early 1966. The background of the language can be found in [Iverson, 1962, McDonnel, 1979, Falkoff and Iverson, 1973]. [Metzger, 1981] talks about the difference in programming methods when using APL as compared to using other languages.

It should be noted that APL was not designed to run on parallel machines. At the time when APL was implemented it was not possible to build (very large) parallel machines. Also, besides the implementation of APL, a time sharing environment was implemented.

Given its early creation date APL remains a highly interesting language. The unusual non-ASCII symbols used for its operators may have prevented the wider use it deserves. The syntax of APL is also peculiar in other ways: for instance, the precedence rule for evaluation of expressions is strictly left-to-right which is very unintuitive and different from all modern languages. This is probably due to the poorly developed parsing technology of the time when APL was defined.

J APL has a descendant called J, where some of the anomalies of APL have been corrected. [Iverson, 1991] For instance, J uses plain symbol names for its operators, as opposed to the special characters of APL.

Code Example 10 This example uses the J language. Input to the J evaluator is indented.

```
   a =. i. 10
   b =. i. 10
   a
0 1 2 3 4 5 6 7 8 9
   c =. a * b
   c
0 1 4 9 16 25 36 49 64 81
   +/ c
285
```

4.2 CM Lisp

When developing the Connection Machine (CM) [Hillis, 1987], Danny Hillis and Guy L. Steele Jr. thought about a programming language for the machine. This language developed into the Connection Machine Lisp (CM Lisp) [Steele Jr. and Hillis, 1986, Hillis, 1987]. CM Lisp never became a commercial language for the Connection Machine, but it is still interesting as a research language. It has been said that it was hard to implement the language efficiently on the CM1. CM Lisp takes inspiration from both APL and FP [Backus, 1978]. The language is an extension to Common Lisp [Steele Jr., 1990], but the carrier language could equally well have been Scheme [Sussman and Steele Jr., 1975].

CM Lisp introduces one additional data type, the *xapping*. A xapping is a combination of an array and a hash table: every element has an index, which can be any lisp object, and a data value, which can also be any lisp object. The index of each element must be unique, which means that a xapping represents a partial function from lisp objects to lisp objects. An example of a xapping (from [Steele Jr. and Hillis, 1986]) is { `sky`$\rightarrow$`blue`, `apple`$\rightarrow$`red`, `grass`$\rightarrow$`green` }. CM Lisp also introduces three new operators: "α," "$\bullet$," and "β." α broadcasts (promotes) scalars into xappings, $\bullet$ "quotes" an object so that it will not be promoted by an α, and β reduces xappings to scalars or performs permutations. Parallel execution comes from doing the elementwise operations on xappings in parallel.

4.3 Miscellaneous Extensions to C

Around 1985/86 quite a few proposals for data parallel extensions of the C programming language appeared.

Vector C Vector C [Li, 1986] introduces a more powerful handling of arrays. Arrays can be manipulated by a special index syntax for scatter/gather etc. The language also introduces a number of intrinsic operations on vectors, all starting with a `@`, like `@.` for inner product and `@>` for maximum in a vector.

Refined C Refined C [Dietz and Klappholz, 1985] is perhaps more aimed at helping the compiler to parallelize the code than to provide actual parallel extensions. (A version for FORTRAN was reported in [Dietz and Klappholz, 1986].)

Parallel-C Parallel-C [Kuehn and Siegel, 1985] is a set of extensions to C where anything can be declared as parallel. Parallel-C has no built in communication. There is supposed to be support for both SIMD and MIMD programming.

MasPar C MasPar C (MPL) has been introduced by the manufacturer MasPar as the C-like language for their parallel machine. [MasPar, 1990] It is quite similar to the New C* language described below.

POMPC The POMPC language is a data parallel extension to C, much like New C* (see section 4.8). POMPC aims at retaining the system-building usability of C, *i.e.* one should be able to write system software for parallel machines in POMPC. As one example one can mention that POMPC supports virtual processors but that it is at the same time possible to access these as they are mapped on the actual hardware, *i.e.* the programmer can use different levels of abstraction for the same data. [Paris, 1992]

4.4 *Lisp

*Lisp [TMC, 1991d, TMC, 1991a] is a programming language for the Connection Machine. *Lisp saw the light of day in March 1986. It is an extension to Common Lisp[Steele Jr., 1990]. The structure of *Lisp very much reflects the PARIS programming model [TMC, 1991b] of the Connection Machine. In this model variables are allocated in *vp-sets*, Virtual Processor-sets. A program executes on a parallel virtual machine that consists of these virtual processors. This virtual machine is emulated on the real parallel hardware. The virtual machine is configured as a grid in any number of dimensions, where the extension of the grid in any direction can be any power of two. Thus, the number of virtual processors can be larger than the number of physical processors in the machine being used. (It cannot, however, be smaller.) The upper limit is given only by the memory available in the physical machine. An apparent advantage is that the same program then can be used on machines of different sizes without any modification.

*Lisp is implemented as a set of powerful macros that "compile" into the intermediate-level language PARIS. There is actually no real compiler. The language lets the programmer define vp-sets as described above.

Code Example 11 This *Lisp function shows how a temporary vp-set and three parallel variables in it can be created.[2] These parallel variables are defined over the virtual processors in the vp-set. The `*all` is a location selector, it makes all locations in the current vp-set active, see Section 2.4.

```
(defun *test (parallel-data-size a-value b-value)
  ;;Type declarations are optional in lisp as well as in *Lisp...
  (declare (type fixnum parallel-data-size)
           (type float a-value b-value))
  ;;Create a one-dimensional vp-set called test-vp-set.
  (let-vp-set (test-vp-set (create-vp-set (list parallel-data-size)))
    ;;Make the vp-set test-vp-set the current one and make all
    ;; processing elements active.
    (*with-vp-set test-vp-set
      (*all
        ;;Create three parallel variables and initiate a!! and b!! by
        ;; broadcasting a-value and b-value.
        (*let ((a!! (!! a-value))
               (b!! (!! b-value))
               c!!)
          ;; perform the elementwise parallel multiplication
          (*set c!! (*!! a!! b!!))
          ;;Sum the elements of c!! and return the sum.
          (*sum c!!))))))
```

*Lisp has special parallel operators for all intrinsic operations on parallel variables, like "`*!!`" for elementwise multiplication. The actual vp-set must be specified by a `(*with-vp-set ...)` statement, which normally would indicate the individual elements of the parallel data being available for scalar operations. Here, however, the semantics is entirely different. The programming model of *Lisp requires a single *current vp-set* to be explicitly set at all times. Any parallel variable referenced, except by communication, must be defined in the current vp-set. `(*with-vp-set ...)` simply sets the current vp-set to that one specified.

[2] By convention all parallel functions that do not return parallel data should have the prefix "`*`," like `*sum` (reduction w.r.t. "+"), and functions that return parallel data should be postfixed with "`!!`," like `+!!` (elementwise "+").

The scope of a variable name in *Lisp is not its vp-set, but rather the same global name space as other variables declared in other vp-sets. One can say that *Lisp is a very explicit programming language; until recently scalars found in parallel expressions, as `a-value` and `b-value` above, had to be broadcast explicitly using `(!! ...)`. The current version of *Lisp (1992) will, however, promote a scalar in a "data parallel context" into a parallel datum defined over the current vp-set. The *Lisp version for the recently released CM5 will even overload ordinary scalar operators, like "`+`", for parallel variables. For the CM5 there will also be a real compiler. *Lisp was originally directed towards the bit-serial architecture of the CM1/CM2 which shows in its very rich type possibilities; one can for instance declare signed integers of any length and specialized floats.

4.5 Old C*

The "Old" C* language [Rose and Steele Jr., 1987] was designed to be an extension to C, such that the extensions had the same flavor as C. It was supposed to be an strict superset of C. Communication was to be done through the use of pointers and pointer arithmetics would be meaningful also in a parallel context. Old C* took a lot from C++, like the new data type `domain`, which was very closely modeled after `class` in C++. The Old C* language was never completely implemented.

4.6 Paralation Lisp

In Paralation Lisp [Sabot, 1988] the single new parallel data type is the *paralation*. A paralation is a vector-like collection of "sites." There are a number of specialized control structures for operations on the parallel data, like "`(elwise (...) ...)`" that performs scalar operations elementwise on paralations. The sites are identified by integers. They represent locations, or processing units. There is also a concept of sites being neighbors, as in a physically distributed multiprocessor. A paralation retains a neighborhood relation for its sites. Communication is done by using `match` and `<-` (move). Communication patterns are called mappings. Communication is quite general. Paralation lisp was defined with no specific computer in mind, but the model executes well on SIMD computers. Paralation Lisp has been implemented on the Connection Machine.

Code Example 12 This example uses Paralation Lisp.

```
(defun test (parallel-data-size a-value b-value)
  ;;Create two paralations and a variable for the result of
  ;; the multiplication.  Initiate the paralations at the same
  ;; time.
  (let ((a-paralation (elwise (make-paralation parallel-data-size)
                              a-value))
        (b-paralation (elwise (make-paralation parallel-data-size)
                              b-value))
        c-paralation)
    ;;Perform the elementwise multiplication operation
    (setf c-paralation (elwise (a-paralation b-paralation)
                               (* a-paralation b-paralation)))
    ;;Return the sum of the c-paralation
    (reduce '+ c-paralation)))
```

4.7 The Evolution of FORTRAN—FORTRAN 8X and FORTRAN 90

Since FORTRAN has such a large user base it has continued to evolve. In the standard proposal FORTRAN 8X, and now in the FORTRAN 90 [Brainerd et al., 1990] standard, a number of "parallel" features are added, like index manipulations of arrays, some intrinsic operators for doing summations, and min/max operations. These intrinsic operators can then be implemented in a parallel way if the target machine is parallel. Arrays can be used in a data-field oriented way, like assignment `A = B`, and elementwise addition `A + B`, given that `A` and `B` are of the same size. There are also operations for referencing parts of arrays, like specifying lower and upper bounds and strides. An example is `A(2:30:4)` which denotes an array formed from every fourth element of `A`, from `A(2)` to `A(30)`. If array indices correspond to processing units, then the use of such expressions usually implies communication. A number of communication patterns are not possible to express in this syntax, which limits the applicability of FORTRAN 90 for parallel programming purposes. For a discussion on FORTRAN 90, see [Albert et al., 1991].

Code Example 13 This example uses CM FORTRAN, a Connection Machine dialect of FORTRAN 90.

```
      PROGRAM EXAMPLE
      INTEGER D
      PARAMETER(N=8)
C Static parallel data
      DIMENSION A(N), B(N), C(N)
C Initiate A and B.
      DATA A / 1, 2, 3, 4, 5, 6, 7, 8 /
      DATA B / 1, 2, 3, 4, 5, 6, 7, 8 /
C Do the elementwise multiplication
      C = A * B
C Print C
      PRINT *, 'C Contains'
      PRINT *, C
C Do the summation
      D = SUM( C )
C Print the sum
      PRINT *, 'The sum is:'
      PRINT *, D

      STOP
      END
```

4.8 The New C*

Since the "Old" C* language (section 4.5) never reached completion, another C derivative was developed for the Connection Machine. This language is the "New" C*. [TMC, 1991c] This is a much less bold language than Old C*. It reflects the underlying hardware of the Connection Machine much more, in the same way as *Lisp.

As in *Lisp, all parallel variables exist in the same name space. Parallel variables are defined over *shapes*. These are a type in New C*. A shape defines the index ranges of a multi-dimensional matrix. Thus, a parallel variable is a matrix of the dimensions specified by its shape. Each matrix element is stored in a different virtual processing unit. Operations over parallel variables are done in parallel, for the different locations in the shape. Just as *Lisp must have a single current vp-set defined at all times, New C* requires a single "current shape" to be set. The underlying machine model is in both cases the same.

New C* is an extension of ANSI C and it conforms well with that language. ANSI C operators are overloaded and are thus syntactically the same for both parallel and scalar expressions. Scalar values in parallel expressions are automatically promoted to parallel values. Communication is done through "left indexing." If `x` is defined over a three-dimensional shape, then its element in processing unit `(17, 42, 4711)` is referenced by `[17][42][4711]x`. This does not infringe on ordinary array references since these use "right indexing." It is thus perfectly possible to have parallel variables whose elements are arrays. If the elements of `x` above are one-dimensional arrays, then we can reference its 13:th element in processing unit `(17, 42, 4711)` as `[17][42][4711]x[13]`. Uniform communication requires a special syntax. All other operations, like reductions and scans, are done with intrinsic operators. New C* does not support the same rich set of data types as *Lisp; instead it is directed towards the data types of ANSI C. This restriction of data types makes it easier to implement the language on the CM5.

Code Example 14 This example uses the New C* language.

```
/* Declare a partially specified shape, this declares the shape's
dimensionality, but not its size. */
shape [][]test_shape;
float test(int vp_set_size, float a_value, float b_value)
{
  /* declare parallel variables and a scalar for the sum */
  float:test_shape a, b, c;
  float sum = 0.0;
  /* allocate the shape of the size we gave */
  allocate_shape( &test_shape, 2, vp_set_size, vp_set_size );
  /* set the current shape to test_shape */
  with( test_shape )
    {
      /* Initiate the parallel variables. */
      a = a_value, b = b_value;
      /* pointwise multiplication */
      c = a * b;
      /* sum */
      sum += c;
    }
  /* deallocate the shape */
  deallocate_shape( &test_shape );
  return( sum );
}
```

5 Classification

In this section we discuss a possible taxonomy of data parallel languages. We first describe and discuss a few possible criteria and how these criteria affect the classification. We also try to assess how important each criterion is. Then we try to classify the languages mentioned above according to the criteria.

5.1 Criteria for Classifying Data Parallel Programming Languages

What properties of data parallel programming languages are especially important and thus suitable as criteria for classification? The criteria that one chooses are of course affected by one's own interests—ours are probably also affected to a large degree. Here we will mainly use the concepts introduced in Section 3.

5.1.1 Target Machine and Target Machine Type

A fairly trivial criterion is the intended target machine of the language. Since this criterion is of practical interest it should be included. It should perhaps be a little more specialized, to both target machine *type* and actual target machine, since how well a language is suited for its type of target machine is much more important than its actual make. Different types of machines are, for instance: vector, data flow, MIMD, and SIMD. Actual machines are, for instance: CRAY, Connection Machine, MasPar, Sequent Symmetry.

5.1.2 Explicit vs. Implicit

A very important property of a data parallel language is whether it is explicit or implicit, *i.e.* how many specialized constructs that have been introduced into the original language.

5.1.3 Control Structures and Intrinsic Operators

The control structures and intrinsic operators are part of the explicit vs. implicit criterion but deserves a little more explanation. A classifying criterion for control structures and intrinsic operators of a data parallel extension is to what extent the carrier language's old ones are applicable to the new parallel data types. An instance of this is overloading of scalar operators to parallel data types. It is also of interest whether the new ones, especially the intrinsic operators, are applicable to the old data types.

5.1.4 Parallel Data

How is parallel data defined in the language, *e.g.* is it considered to be in the form of distinct parallel entities or contained in objects with a shape? Another interesting criterion is what restrictions the language puts on the elements in parallel data. Must the locations always be positive integers? Can any object allowed in the language be used as a label for a processing unit?

A final interesting property of shapes is whether they can be recursive, *i.e.* whether a shape can contain other shapes.

5.1.5 Miscellaneous

A lot of other criteria could probably be used to describe and classify data parallel programming languages, but only one more will be mentioned. That is whether the syntax of the language reflects the cost of the operations being carried out by a particular syntactical construct. Should a language hide the complexity and possibly mislead the programmer about the efficiency of writing a program in a certain way? This is perhaps not as important a criterion as the others mentioned above. We believe that a data parallel language first and foremost should be expressive, easy to use, and portable. If the language is portable it is, however, very hard to predict the efficiency of the language unless we know the architecture of the actual hardware and how the language constructs are mapped to that hardware.

5.2 Classification of Existing Languages

Let us now classify some of the languages mentioned in Section 4 according to the criteria stated above. The languages are APL, CM Lisp, *Lisp, Old C*, Paralation Lisp, FORTRAN 90, and New C*.

Target Machine and Target Machine Type APL was first implemented on IBM systems and has since then been implemented on a lot of machine types. Currently we do not know of any complete APL implementation on a parallel computer. A small version on APL, called AAPL, was used as application language on the AAP2 computer of NTT [Kondo et al., 1986]. CM Lisp has only been simulated on conventional computers. *Lisp, Old C*, Paralation Lisp, CM FORTRAN (a modified subset of FORTRAN 90), and New C* all run on the Connection Machine. FORTRAN 90 has been, or is being implemented on many vector machines like CRAY.

Explicit versus Implicit APL, CM Lisp, and Paralation Lisp have special operators that manipulate parallel data. They are thus rather explicit about handling parallel data. *Lisp, Old C*, and New C* are all intended to run on the same hardware and are all explicit in their way of expressing parallelism. Of these, New C* is slightly less explicit since it has nicely overloaded operators. FORTRAN 90 has no particular parallel data type. The possible parallelism comes from operating on arrays in parallel. Since it is up to the compiler to decide whether to implement arrays conventionally or in parallel, we classify FORTRAN 90 as implicit.

Control Structures and Intrinsic Operators APL, as far as control structures and intrinsic operators goes, has to be put in a class of its own, since it has a syntax all of its own. CM Lisp and Paralation Lisp both have very special ways of mapping operations onto the parallel data; CM-Lisp uses α and β and Paralation Lisp uses `(elwise (...) ...)` mentioned in sections 4.6 and 4.2, respectively. *Lisp, Old C*, and New C* all have control structures close to those in the original languages. In *Lisp the if-statements has two new data parallel versions, `(*if ...)` and `(if!! ...)` The former is a location selector that turns on and off locations depending on the outcome of a parallel test: the specified parallel operations for each branch are then performed in a certain location depending on the local outcome of the test. The latter selects locations and directs execution in the same way, but is also a function that returns a parallel result. These languages have new control structures for selecting the current parallel datum, like `(*with-vp-set ...)` and `with { ... }` mentioned in sections 4.4 and 4.8, respectively. Old C* builds on thoughts from C++ and New C* is well integrated with ANSI C. The intrinsic operators of *Lisp are modeled closely after those in Common Lisp, whereas New and Old C* have quite a few new intrinsic operators, especially for reduction operations. FORTRAN 90 has many new features but is still rather restrictive about their use. It has a lot of intrinsic operators, *e.g.* for sums and maximums.

Parallel Data Both APL and FORTRAN 90 have arrays as carriers of parallel data. Array elements are referenced with an integer index. CM Lisp and Paralation Lisp introduces new powerful concepts for handling the data. In CM Lisp the xapping can have any lisp atom as index as long as it is unique. Paralation Lisp implicitly introduces an integer index on all paralations, and this index has to be used. *Lisp, Old C* and New C* have a matrix-like way of handling the parallel data. The elements of the parallel data are referenced much like arrays but with a different syntax, which enables the explicit use of both conventionally stored arrays and parallel data in the same program. In *Lisp and New C* the current vp-set must always be specified. This effectively prohibits elementwise expressions with variables from different vp-sets. To perform operations with variables from different vp-sets, they must all first be cast into a single vp-set. This means mapping virtual processors from the original vp-sets to virtual processors in the common vp-set. Then the operations can be performed, and possibly some results will finally be mapped back into some of the original vp-sets.

Miscellaneous Since all languages, except possibly CM Lisp, have intrinsic operators where the complexity is very dependent on the implementation on the actual machine, it is rather hard to assess the efficiency of a program just from looking at it. APL is rumored to be rather hard to write efficient programs in. Both CM Lisp and Paralation Lisp are so different from traditional programming languages that it might be hard to assess them using "rules of thumb" from traditional sequential programming.

The following table is an attempt to give a comprehensive view of the immediately comparable classification criteria of the different languages.

language	**target machine type**	**explicit/implicit**	**parallel data**
APL	no special	explicit	array
CM Lisp	SIMD (CM-1)	explicit	xapping
*Lisp	SIMD (CM-2)	explicit	pvar (distr. matrix)
Old C*	SIMD (CM-2)	explicit	pvar
Paralation Lisp	SIMD	explicit	paralation
FORTRAN 90	SIMD/vector	implicit	array
New C*	SIMD (CM-2)	explicit	pvar

6 Proposal for a New Model

As mentioned previously, data parallel programming languages have been designed either from a hardware point of view or in order to be easily vectorized or parallelized. Neither of these viewpoints are actually very good if the aim is to design a portable and expressive data parallel language that is easy to use. We believe that a data parallel programming language should express a *data parallel paradigm* and not necessarily be a model of *SIMD computation*. In fact any data parallel programming language can be implemented and executed on any hardware platform. Data parallelism is a programming paradigm suitable for certain applications—especially applications handling large amounts of data. So, we henceforth do *not* equal data parallel computation with vector or SIMD computation. (Needless to say, these architectures sometimes *do* implement data parallel processing very efficiently.) A data parallel language should have operators with clear semantics that allow them to be executed in parallel, but that they should not model parallel execution of a specific architecture. We believe there is a distinction here, and a vital one at that!

Existing data parallel languages have many built-in primitives. We do not believe that a large number of primitives necessarily gives a language large expressive power. On the contrary, a large number of primitives complicates the syntax and it becomes hard to learn them all. A similar tendency has been seen for sequential languages, as observed in [Backus, 1978]. Along the same lines, with inspiration from functional programming, we would like to propose a view of data parallelism that we believe can lead to data parallel languages with a larger degree of generality, clarity and semantic soundness. For a more elaborate description of the below, see [Hammarlund and Lisper, 1993].

6.1 Parallel Data as Functions

We define parallel data as *parallel data fields.* A parallel data field is a *function* from a *index domain* I to a *data domain* A, *i.e.* an entity of type $I \to A$. Both I and A are sets with elements of specified types.[3] A "shape" in existing data parallel languages is then a special case of index domain. With a suitable syntax for function definitions, more general index domains can be expressed than is possible in existing data parallel languages. An example of practical interest is sparse index domains. Our view of data parallelism is close to the interpretation of arrays in the language Crystal [Yang and Choo, 1992].

In a functional language there is no semantic difference between parallel data fields and program code. Whether to consider a function as a parallel data field or as a program is merely a matter of implementation. An implementation of a parallel data field is essentially *a representation as a distributed table*, where each element in the index domain is mapped to a physical processing unit. Thus, a function must have a finite index domain to be possible to represent in this way. Most functional languages have no particular means to restrict the domain of functions in this way: functions are usually defined for unbounded domains such as integers, or lists. A data parallel functional language must therefore have some additional constructs to restrict the index domain. Crystal has some simple primitives for this: there, index domains can be constructed from contiguous integer intervals, cartesian product, direct sum, and predicates selecting elements from already defined domains. We believe that more general ways of defining finite index domains are possible and desirable.

An interesting possibility is lazy evaluation of parallel data fields. That would mean generating only the portion of the distributed table that is actually needed. We can thus have potentially infinite parallel data fields. The use of "lazy data fields" will require some kind of dynamic mapping from the index domain to physical processing units.

Another possibility is currying of multi-dimensional parallel data fields. For example, a three-dimensional parallel data field $(I_1 \times I_2 \times I_3) \to A$ can be seen as a curried data field $I_1 \to (I_2 \to (I_3 \to A))$. This can be considered a parallel data field *containing* parallel data fields.

Note that a table representing a parallel data field need not be stored in a distributed fashion. It could well be stored in local memory and accessed sequentially. In that case a conventional array implementation results. Currying of data fields could then be interpreted as selecting

[3] Note that we can allow a polymorphic type system, with hierarchical types much like in Common Lisp [Steele Jr., 1990]. Then operators will only have to be applicable to that subtree of the type hierarchy.

some dimensions to be stored distributed over different processing units and other dimensions to be stored locally as arrays. Since the curried and the uncurried views of a function are equivalent, this yields interesting possibilities for automatic conversion between totally distributed representations and representations that are partially distributed and partially conventional.

6.2 Elementwise Application of Functions

Elementwise application of scalar operations is simply functional composition in our model. A function $g : A \rightarrow B$ applied elementwise to a parallel data field $f : I \rightarrow A$, yields a new parallel data field $g \circ f : I \rightarrow B$. In this way we avoid the introduction of a parallel version of g.

6.3 Communication and Selection

Communication is modeled by index domain transformations. A mapping $\pi : I_2 \rightarrow I_1$ transforms a parallel data field $f : I_1 \rightarrow A$ into a new parallel data field $f \circ \pi : I_2 \rightarrow A$. If each index i in I_1 is mapped to processing unit p_i and each index j in I_2 is mapped to processing unit q_j, then π can be interpreted as a concurrent read from $p_{\pi(j)}$ to q_j, for all $j \in I_2$.

Send operations result from a "backwards" interpretation of index domain transformations. A reversed direction mapping $\pi_2 : I_1 \rightarrow I_2$ can be seen as a concurrent send from p_i to $q_{\pi_2(i)}$, for all $i \in I_1$. A parallel data field $f : I_1 \rightarrow A$ is then transformed into a new parallel data field $g : \pi_2(I_1) \rightarrow A$ given by $g(j) = f(\pi_2^{-1}(j))$ for all $j \in \pi_2(I_1)$. If π_2 is not injective, then $\pi_2^{-1}(j)$ is not uniquely defined for some j which corresponds to a write conflict. A possible way to resolve the conflict is to apply some reducing function to all elements mapped to the same point. In this way combining communication and segmented scans can be expressed. The segments are then simply the partitions of I_1 with respect to π_2.

Selection is referencing a subset or a particular element of the parallel data field. Selecting a subset of a parallel data field is simply function restriction to a subset of the index domain. Selection of a particular element i of a parallel data field f is just function application: $f(i)$.

7 Discussion

It can be argued that a general high level programming language based on our view of data parallelism may be very hard to implement efficiently. With the present state of affairs this is certainly true. We would, though, like to argue that:

1. There is a comparatively large research initiative in data parallel implementations of functional programs. Related to our proposed model work is being done on compiling composition languages [Budd, 1988b] and collection oriented languages [Blelloch and Sabot, 1990]. We believe that high-level programs can be compiled to code that is only a constant factor worse than the corresponding "hand coded" program.

2. The higher level of abstraction of high level languages makes them less dependent on a certain architecture. Coding a program close to a certain architecture will most likely make it run worse on a different architecture. If both portable and efficient code is desired, then it seems that high-level languages with advanced compilation methods is the only way to go.

Work is currently done on compiling data parallel languages for vector machines [Budd, 1988a] and for SIMD computers [Gilbert and Schreiber, 1991]. A high-level language compiler can both use algorithmical skills [Hillis and Steele Jr., 1986] and knowledge about how a parallel data structure can be optimally mapped onto a machine [Knobe et al., 1990]. For parallel architectures the communication problem is becoming more understood [Saltz et al., 1991, Ho, 1991, Edelman, 1990, Papadimitriou and Sipser, 1984]. We hope to be able to include some of these results in our research on implementations of data parallel languages.

Finally it should be noted that there are other research efforts working towards much the same goals [Rice, 1990, Galil and Paul, 1983, Yang and Choo, 1992].

8 Conclusions

We introduced the concept of data parallel computation and we showed a few examples of basic computations like elementwise operations, communication, and reductions. We continued to discuss the possible opportunities and problems of data parallel programming versus sequential programming. We exemplified some possible pitfalls and explained how these can be bypassed.

We argue that the data parallel computation paradigm is a strong and useful one. Data parallel programming remedies some of the problems of sequential programming but also introduces some new ones. These problems can be avoided if the semantics for the data parallel constructs is sufficiently abstract. In the survey of data parallel programming we discussed some already implemented data parallel languages. Many of these languages were designed with with a strong consideration of the target hardware: thus, they lack a sound theoretical foundation. We found, though, that they are quite effective for their target architectures. By picking a few criteria that describes data parallel languages we then proceeded to classify these languages. Then, finally, we presented our view of data parallelism. The implementation issues of languages based on this model are not discussed in this report, except briefly in section 7.

References

[Albert et al., 1991] Albert E, Lukas J. D, & Steele Jr. G. L, (1991). Data Parallel Computers and the FORALL Statement. *Journal of Parallel and Distributed Computing*, 13:185–192.

[Backus, 1978] Backus J, (1978). Can Programming Be Liberated from the von Neuman Style? A Functional Style and Its Algebra of Programs. *Comm. ACM*, 21(8):613–641.

[Blelloch and Sabot, 1990] Blelloch G. E & Sabot G. W, (1990). Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8:119–134.

[Brainerd et al., 1990] Brainerd W. S, Goldberg C. H, & Adams J. C, (1990). *Programmer's Guide to FORTRAN 90.* Programming Languages. McGraw-Hill.

[Budd, 1988a] Budd T. A, (1988). A New Approach to Vector Code Generation for Applicative Languages. Technical report, Department of Computer Science, Oregon State University, Corvallis, Oregon 97331.

[Budd, 1988b] Budd T. A, (1988). Composition and Compilation in Functional Programming Languages. Technical report, Department of Computer Science, Oregon State University, Corvallis, Oregon 97331.

[Dietz and Klappholz, 1985] Dietz H & Klappholz D, (1985). Refined C: a sequential language for parallel programming. In DeGroot D (ed), *Proc. 1985 International Conference on Parallel Processing*, pp 442–449. IEEE Computer Society, August 1985.

[Dietz and Klappholz, 1986] Dietz H & Klappholz D, (1986). Refined FORTRAN: another sequential language for parallel programming. In Hwang K, Jacobs S. M, & Swartzlander E. E (eds), *Proc. 1986 International Conference on Parallel Processing*, pp 184–191. IEEE Computer Society, August 1986.

[Dijkstra, 1976] Dijkstra E. W, (1976). *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J.

[Edelman, 1990] Edelman A, (1990). Optimal Matrix Transposition and Bit Reversal on Hypercubes: All-To-All Personalized Communication, 1990. Personal communication.

[Falkoff and Iverson, 1973] Falkoff A & Iverson K, (1973). The Design of APL. *IBM Journal of Research and Development*, pp 324–333.

[Galil and Paul, 1983] Galil Z & Paul W. J, (1983). An efficient General-Purpose Parallel Computer. *Journal of the Association for Computing Machinery*, 30(2):360–387.

[Gilbert and Schreiber, 1991] Gilbert J. R & Schreiber R, (1991). Optimal Expression Evaluation for Data Parallel Architectures. *Journal of Parallel and Distributed Computing*, 13:58–64.

[Gries, 1978] Gries D, (1978). The Multiple Assignment Statement. *IEEE Trans. Software Eng.*, SE-4(2):89–93.

[Hammarlund and Lisper, 1993] Hammarlund P & Lisper B, (1993). On the relation between functional and data parallel programming languages. In *Proc. of the Sixth Conference on Functional Programming Languages and Computer Architecture*, pp 210–222. ACM Press, june 1993.

[Hillis and Steele Jr., 1986] Hillis W. D & Steele Jr. G. L, (1986). Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183.

[Hillis, 1987] Hillis W. D, (1987). The Connection Machine. *Scientific American*, 256:108–115.

[Ho, 1991] Ho C.-T, (1991). Optimal Broadcasting on SIMD Hypercubes Without Indirect Addressing Capability. *Journal of Parallel and Distributed Computing*, 13:246–255.

[Iverson, 1962] Iverson K. E, (1962). *A Programming Language.* Wiley, New York.

[Iverson, 1991] Iverson K. E, (1991). *Programming in J.* Iverson Software Inc., Toronto.

[Knobe et al., 1990] Knobe K, Lukas J. D, & Steele Jr. G. L, (1990). Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118.

[Kogge and Stone, 1973] Kogge P. M & Stone H. S, (1973). A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Trans. Comput.*, C-22:786–793.

[Kondo et al., 1986] Kondo T, Tsuchiya T, Kitamura Y, Sugiyama Y, Kimura T, & Nakashima T, (1986). Pseudo MIMD Array Processor — AAP2. *IEEE.*

[Kuehn and Siegel, 1985] Kuehn J. T & Siegel H. J, (1985). Extensions to the C Programming Language for SIMD/MIMD Parallelism. In Degroot D (ed), *Proc. 1985 International Conference on Parallel Processing*, pp 232–235. IEEE Computer Society, August 1985.

[Li, 1986] Li K.-C, (1986). A Note on the Vector C Language. *ACM SIGPLAN Notices*, 21(1).

[MasPar, 1990] MasPar, (1990). MasPar data-parallel programming languages. MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California 94086.

[McDonnel, 1979] McDonnel E, (1979). The Socio-Technical Beginnings of APL. *APL Quote Quad*, pp 13–18.

[Metzger, 1981] Metzger R. C, (1981). APL Thinking, Finding Array-Oriented Solutions. *ACM*, pp 212–218.

[Papadimitriou and Sipser, 1984] Papadimitriou C. H & Sipser M, (1984). Communication Complexity. *Journal of Computer and System Sciences*, 28:260–269.

[Paris, 1992] Paris N, (1992). The POMPC data parallel language. Presented at the first European Connection Machine Workshop, Wuppertal., June 1992.

[Rice, 1990] Rice M. D, (1990). Semantics for Data Parallel Computation. *International Journal for Parallel Programming*, 19(6):477–509.

[Rose and Steele Jr., 1987] Rose J. R & Steele Jr. G. L, (1987). C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the 1987 Second International Conference on Supercomputing*, volume II, pp 2–16. International Supercomputing Institute, 1987.

[Sabot, 1988] Sabot G. W, (1988). Paralation Lisp Reference Manual. Technical Report PL87–11, Thinking Machines Corporation, Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142.

[Saltz et al., 1991] Saltz J, Petiton S, Berryman H, & Rifkin A, (1991). Performance Effects of Irregular Communication Patterns on Massively Parallel Multiprocessors. *Journal of Parallel and Distributed Computing*, 13:202–212.

[Steele Jr. and Hillis, 1986] Steele Jr. G. L & Hillis W. D, (1986). Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing. Technical Report PL86–2, Thinking Machine Corporation, Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142.

[Steele Jr., 1990] Steele Jr. G. L, (1990). *Common Lisp the Language II.* Digital Press, 12 Crosby Drive, Bedford, Massachusetts 01730, second edition.

[Sussman and Steele Jr., 1975] Sussman G. J & Steele Jr. G. L, (1975). Scheme: An interpreter for an extended lambda calculus. AI Memo 349, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts.

[TMC, 1989] TMC, Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142, (1989). *Connection Machine Model CM-2 Technical Summary*, 1989.

[TMC, 1991a] TMC, Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142, (1991). *Connection Machine: *Lisp Dictionary*, 6.1 edition, 1991.

[TMC, 1991b] TMC, Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142, (1991). *Connection Machine: Parallel Instruction Set (Paris)*, 6.1 edition, 1991.

[TMC, 1991c] TMC, Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142, (1991). *Connection Machine: Programming in C**, 6.1 edition, 1991.

[TMC, 1991d] TMC, Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142, (1991). *Connection Machine: Programming in *Lisp*, 6.1 edition, 1991.

[Yang and Choo, 1992] Yang J. A & Choo Y, (1992). Data fields as parallel programs. In *Proceedings of the Second International Workshop on Array Structures*, Montreal, Canada, June/July 1992.