I.J.Modern Education and Computer Science, **2012, 1, 1-11**
Published Online February 2012 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2012.01.01

Element-Based Computational Model

Conrad Mueller
School of Electrical and Information Engineering, The University of the Witwatersrand and School of IT Monash South Africa, Johannesburg
Email: cmueller@acm.org

***Abstract*—A variation on the data-flow model is proposed to use for developing parallel architectures. While the model is a data driven model it has significant differences to the data-flow model. The proposed model has an evaluation cycle of processing elements (encapsulated data) that is similar to the instruction cycle of the von Neumann model. The elements contain the information required to process them. The model is inherently parallel. An emulation of the model has been implemented. The objective of this paper is to motivate support for taking the research further. Using matrix multiplication as a case study, the element/data-flow based model is compared with the instruction-based model. This is done using complexity analysis followed by empirical testing to verify this analysis. The positive results are given as motivation for the research to be taken to the next stage - that is, implementing the model using FPGAs.**

***Index Terms*—Computational Model, Data-Flow, Computer Architecture, Parallel Architecture.**

I. Introduction

The following two quotes are the motivation for the research.

"For more than 30 years, researchers and designers have predicted the end of uniprocessors and their dominance by multiprocessors. During this time period the rise of microprocessors and their performance growth has largely limited the role of multiprocessors to limited market segments. In 2006, we are clearly at an inflection point where multiprocessors and thread-level parallelism will play a greater role..." [1]

"Since real world applications are naturally parallel and hardware is naturally parallel, what we need is a programming model, system software, and a supporting architecture that are naturally parallel. Researchers have the rare opportunity to re-invent these cornerstones of computing, provided they simplify the efficient programming of highly parallel systems." [2]

Given the need for parallelism, is it not worth looking at previous attempts at parallel models? In the 1970s, the data-flow model was explored with some limited success [3]. This paper explores how to build on some of the concepts of the data-flow model. This new data-driven model has significant differences from the previous data-flow models, as described below, as well as being inherently parallel, which simplifies parallel programming.

Table 1: Comparison between Data-flow and Element

	Data-flow	Element-based
program	cyclic graph	acyclic graph
graph	finite	infinite
edge	pointer	element
node	instruction	relation
values	memory	elements
computation	firing instruction	processing element
current state	values in nodes	active elements
result of op	a value	a new element
selection	instructions	partial functions

As Hennessy [4] has pointed out, implementing a new architecture requires considerable resources. These are unavailable to us. The objective of this paper is to provide some evidence that it is worthwhile taking this research further, in the hope that others may see some potential in collaborating in this research. The method chosen to provide this evidence is to experiment with an example program and assess how the new model is likely to compare with the predominant instruction-based model.

The model is built on theory relating to functions and sets that forms both the basis of the model and a language to express programs. Developing the language is a research project on its own and its justification depends on acceptance of the model. For this reason the focus has been on the model. Understanding programs written in the language only requires an understanding of simple algebraic mathematical expressions. That is sufficient for this paper.

The question is how best to assess the potential of a model given minimal resources. The approach chosen is to compare the model with the instruction-based model in two ways. The first comparison is the cost of data accesses per instruction versus per element processed. By studying the execution loops of each, an argument is made that processing an element is at least as efficient as executing an instruction. This does not give an indication of how they compare when doing real computations and in particular parallel computations. An example is used to explore this aspect, first by comparing the number of instructions to the number of elements processed and then by comparing the instruction-based run-time performance with that of the emulation of an element model.

The paper reports experimental results of simulating some of the behavior of this model. These results suggest that such an architecture's performance scales well with increases in processors. An MPI [5] emulation is used to compute a matrix multiplication with increasing sizes and increasing numbers of processors. Based on given assumptions, an argument is made that an architecture based on this model scales linearly for the example.

The rest of the paper is broken up into three sections. The first given a brief description of the model and compares it with other architectures. The next section describes the research method. The final section evaluates the research and considers how to take the research further.

II. Model of Computation

A. Basics

In many ways our model is similar to the von Neumann model; however, instead of processing instructions, it processes elements. The elements consist of information that uniquely conveys the meaning of the data as well as a value. This information is used to determine how the element is processed. Processing an element results in zero or more elements being created. The program execution completes when there are no more elements to process. A comparison between the instruction model and element model execution cycle is given below in Table 2.

Table 2: Comparison of Execution Cycles

Instruction-model	Element-model
get instruction	get element
get data	get relation
perform instruction	apply relation
store result in memory	push element(s) on queue

Without going into too much detail, given an element *b=5*, a relation *a=-b* and the steps above the result is the element *a=-5* being created. The question arises as to how one deals with an expression of the form *a=b+c* as the model only processes elements. This relation can be viewed as the operation + being applied to an element that is a tuple, i.e. *(b,c)*. The way around this is that the elements *b* and *c* are used to create an element, say *t*, to which plus can be applied to create *a*. Thus the expression *a=b+c* is broken up into:

$$t = \ltimes b \qquad t = \rtimes c \qquad a = +t \tag{1}$$

where the operation is $\ltimes$ to create the zeroth part of the tuple and $\rtimes$ is to create the first part of the tuple. Alternatively it may be expressed as:

$$b \overset{\ltimes}{\rightarrow} t \qquad c \overset{\rtimes}{\rightarrow} t \qquad t \overset{+}{\rightarrow} a \tag{2}$$

Table 3 illustrates the computation given for c=3, b=5.

Table 3:Evaluation of *a=b+c*

Element	Step	Comment
$c = 3$	1	pop element
	2	apply $c \overset{\rtimes}{\rightarrow} t$
		t not found in hash table
	3	$t = 3$ in hash table
$b = 5$	1	pop element
	2	apply $b \overset{\ltimes}{\rightarrow} t$
		t found in hash table
		joint $= 5$ joined with $t = 3$
	3	$t = (5,3)$ push on queue
$t = (5,3)$	1	popped off the stack
	2	relation $t \overset{+}{\rightarrow} a$ is used
	3	$a = +(5,3) = 8$
		$a = 8$ push on queue
$a = 8$	1	popped off the stack
	2	no relation
	3	no element created

A similar process allows one to decompose any number of binary or unary operations into a number of expressions consisting of one unary operation.

Describing any meaningful computation requires selection. The *if* operation is used to express selection. Here is an example of how it is used.

$$\begin{aligned} z &= x - y \text{ if } x > y \\ &= x \text{ if } x \leq y \end{aligned} \tag{3}$$

The trick used here is to treat *if* as an operation. As in the C programming language, *?* is used to denote *if*. We can express the first line as a graph, illustrated in Figure 1 and as a sequence of expressions and relations in Table 4. The evaluation of these relations for *x=3* and *y=10* is shown in Table 5. Computing the expression $z = x$ **if** $x \leq y$ results in the creation of the element $z = 3$.

Finally to provide a complete computational model, repetition needs to be addressed. The following example of summation illustrates how to achieve this.

$$\begin{aligned} s_0 &= 0 \\ s_{i+1} &= s_i + n_i \\ a &= s_i \text{ if } n_i = \textit{ end of line} \end{aligned} \tag{4}$$

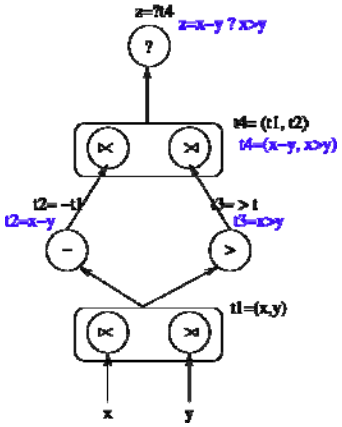

Figure 1:Graph of the expression $z = x\text{-}y$ *if* $x>y$

Table 4: Primitive Expressions for $z = x\text{-}y$ **if** $x>y$

expressions	*relations*
$t1 = \ltimes x$	$x \xrightarrow{\ltimes} t1$
$t1 = \rtimes y$	$y \xrightarrow{\rtimes} t1$
$t2 = -t1$	$t1 \xrightarrow{-} t2$
$t3 = > t1$	$t1 \xrightarrow{>} t3$
$t4 = \ltimes t2$	$t2 \xrightarrow{\ltimes} t4$
$t4 = \rtimes t3$	$t3 \xrightarrow{\rtimes} t4$
$z = ?\, t4$	$t4 \xrightarrow{?} z$

Table 5: Evaluation of $z = x\text{-}y$ *if* $x>y$

Element	Relation	Action	Result
$x = 3$	$t = \ltimes x$	*hash:* $t1 = 3$	
$y = 10$	$t = \rtimes y$	*join:* $t1 = 10$	$t1 = (3, 10)$
$t1 = (3, 10)$	$t2 = -t1$	*compute:* $t2$	$t2 = -7$
	$t3 = > t1$	*compute:* $t3$	$t3 =$ *false*
$t2 = -7$	$t4 = \ltimes t2$	*hash:* $t4 = -7$	
$t3 = f$	$t4 = \rtimes t3$	*join:* $t4 = f$	$t4 = (-7, f)$
$t4 = (-7, f)$	$z = ?\, t4$	*compute:* z	**none**

Table 6: Primitive expressions for $s_{i+1} = s_i + n_i$

expression	*relation*
$t1 = \ltimes s$	$s \xrightarrow{\ltimes} t1$
$t1 = \rtimes n$	$n \xrightarrow{\rtimes} t1$
$t2 = +t1$	$t1 \xrightarrow{+} t2$
$s = [+1]_0\, t2$	$t2 \xrightarrow{[+1]_0} s$

where $[+1]_0$ indicates add one to index zero.[1]

The expression $s_{i+1} = s_i + n_i$ is used to describe how iteration is achieved. Indices are introduced that do not have bounds. This expression is an infinite graph as shown in Figure 2.

[1] For simplicity we have not gone into the detail of how any number of indices is handled. Note that the index operation is explicitly specified and thus the index does not need to be captured as part of a relation.

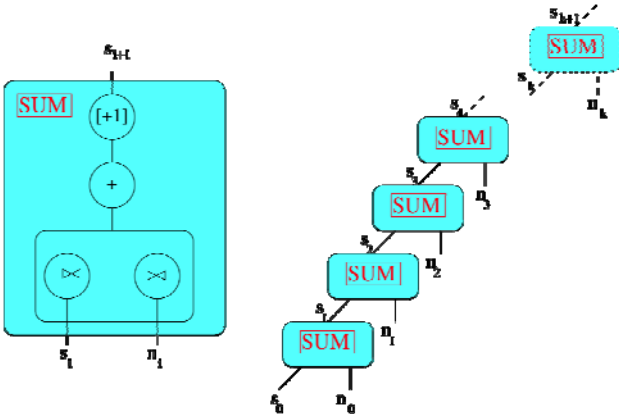

Figure 2: Graph of the expression $s_{i+1} = s_i + n_i$

An evaluation of these relations for $s_0=0$, $n_0=3$, $n_1=7$ and $n_2=13$ is shown in Table 7.

Table 7: Evaluation of Summation

Element	Expression	Result
$s_0 = 0$	$t = \ltimes s$	$t1_0 = 0$
$n_0 = 3$	$t = \rtimes n$	$t1_0 = 3, t1_0 = (0, 3)$
$t1_0 = (0, 3)$	$t2 = +t1$	$t2_0 = 3$
$t2_0 = 3$	$s = [+1]_0\, t2$	$s_1 = 3$
$s_1 = 3$	$t = \ltimes s$	$t1_1 = 3$
$n_1 = 7$	$t = \rtimes n$	$t1_1 = 7, t1_1 = (3, 7)$
$t1_1 = (3, 7)$	$t2 = +t1$	$t2_1 = 10$
$t2_1 = 10$	$s = [+1]_0\, t2$	$s_2 = 10$
$s_2 = 10$	$t = \ltimes s$	$t1_2 = 10$
$n_2 = 13$	$t = \rtimes n$	$t1_2 = 13, t1_2 = (10, 13)$
$t1_2 = (10, 13)$	$t2 = +t1$	$t2_2 = 23$
$t2_2 = 23$	$s = [+1]_0\, t2$	$s_3 = 23$

The Figure 3 shows the structure of a simple implementation of the model consisting of one processor. This implementation consists of: a queue of unprocessed elements that are waiting to be processed, the table relations and hash table. Elements are popped off the queue, processed *using* the relations – where the relation has a tuple operation the tuple is formed using the hash table, and finally the newly created elements are added to the queue.

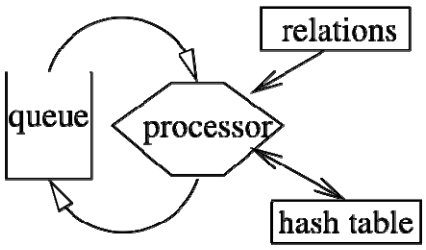

Figure 3: Structure of Element Model

The processor takes elements off the queue and loops through the relations indexed by the element name. The processor applies the operation in each relation to the values of the element. If the *operation* is defined, a new element is created with the range element name in the

relation and the values or indices resulting from applying the operation. The newly created element is pushed onto the stack. The process continues until there are no more elements to process.

There are three classes of operations: those that result in a new value being computed; those that result in a new index being computed and those that compute a tuple. The paper does not go into the detail of all these operations. The only one requiring a little more detail are the operations that create tuples. The two elements used to create a tuple can come in any order. The first to arrive of the two elements that form a tuple is inserted in the hash table. The second element to arrive is then matched and the tuple is formed. The element name and the indices are used to match the two elements that form the tuple.

The model is inherently parallel as any processor that has the relations to process that element can process any element[2]. Thus the element in the unprocessed queue can be distributed across any number of processors.

B. Comparison with other architectures

The element-based architecture has no instructions. The only other models that do not use instructions are theoretical models like the Turing Machine. Data-flow architectures do not have instructions executed in a sequential order like the von Neumann. However, the nodes are typically referred to as firing instructions. Even so, data-flow architectures come close in terms of shifting the emphasis from executing instructions to processing data[6,7,8].

As with data-flow, element-based model programs can be viewed as graphs. The big difference is that the data-flow graphs are finite and allow for cycles, whereas in the element model programs, the graphs can be infinite and acyclic. A data-flow program is stored as a graph with pointers to the nodes in memory, restricting the graph to being finite and thus cyclic. A data-flow program cannot be represented by an infinite graph as can be done with the element-based approach. The element model achieves this by expressing the graph using classes of relations that represent infinitely many nodes and edges and not representing the graph as nodes and pointers in memory.

The unique aspect of this model is that values are not referenced by memory addresses, but rather, the semantic information required to process the elements is kept together with values. The processor determines how to process the value based on the element name and indices. In the von Neumann model it is an instruction that determines how the value will be processed. In the data-flow it is the instruction that gets fired when the required values arrive at a node.

Like the von Neumann model, the element model has a simple three step execution cycle but instead of executing the next instruction, the next element on the queue is processed. There is no concept of the program counter and hence the state of the program execution is determined by active elements that are on the queue of elements waiting to be processed. This contrasts with the von Neumann model, where the program counter and the contents of memory capture the state, or the data-flow model where the state is determined by the state of the nodes.

The data-flow model allows for parallelism in that any number of nodes can fire simultaneously. However there are a number of limitations. The cycles in the graph limit the degree to which available values can be processed. Either starting the next cycle has to wait for the previous cycle to finish or there has to be some mechanism for differentiating between cycles. The nodes are stored in memory that restricts the evaluation of values to take place in the processor where the nodes are stored. The element model is able to process all the active elements in any order and any number at the same time, limited only by the number of processors.

Being able to process elements in this way facilitates parallelism for a number of reasons. Processing an element is independent of other active elements and hence of the state of the program, thus any processor that has access to the relations associated with an element can process that element. The exception is forming tuples that requires a mechanism to handle that the two components of the tuple need to access the same hash table. It does come at a cost and that is an aspect that this research attempts to evaluate.

The element model is a functional one, yet different from current models. Instead of expressing and evaluating a program as a function that gets applied to values, a program is a set of algebraic expressions of relations between sets of elements. Such a relation defines the mapping between the elements of the two sets. A relation and an element in its domain are used to create an element in the range. Thus a relation, such as

$$a = f(b), \tag{5}$$

and an element in the domain, e.g.

$$b=(5,3), \tag{6}$$

are enough to determine the element

$$a=8. \tag{7}$$

An element consists of an identifier (name and indices) and a value. The element name is used to establish for which relations the element is a member of its domain. When a new element is created, the relation determines what the identifier is and what operation has to be applied to determine the value of the new element.

Identifiers are used in a different way. Most languages typically express a function such as square something like *square(x) = x*x*.

[2] Except in the case where a tuple operation is involved, where the semantic information of the element is used to redirect this operation to where its partner will be directed to.

In this example the variable identifier *x* has no significance other than that it represents the value passed to the function. The variable *x* can be used in the definition of other functions and using the same variable does not imply any relationship between these functions and the function *square*. An element example of a relation is

$$\text{volume} = \text{area} * \text{height}. \quad (8)$$

Here the meaning of *area* is not confined to this relationship. Hence using the same identifier in another relation relates the two relations. For example if there is another relation

$$\text{area} = \text{length} * \text{breadth}, \quad (9)$$

then *area* refers to the same entity in both relations. Whereas the functional approach sees the function *square* as the mapping

$$\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R},$$

the element-based relation describing *volume* is viewed as the mapping

$$\mathit{area} \times \mathit{height} \rightarrow \mathit{volume}. \quad (10)$$

The variable *x* has no significance once the function *square* has been computed. The element *area* has a life independent of the relation that created it and its meaning holds for the entire computation.

Since the meaning of an element holds for the entire computation, the order in which it is processed is not important. This property enables elements to be processed in any order as well as on different processors, facilitating parallel processing. This property also helps with programming. The programmer does not need to have to design the order in which the execution takes place and, more important, does not need to design the coordination between the processors. Since the meanings of elements are static, relations also are. This is unlike instruction-based programming where the programmer needs to take into account that a statement has different meanings as different values are iterated through it.

Given that in any program one is limited to a finite number of identifiers, this suggests that one is limited to a fixed number of elements and computations. Both instructional and functional models get around this by reusing identifiers. The element-based model overcomes this limitation by having element identifiers consisting of a name and indices. For practical purposes, this enables infinitely many unique element identifiers. By using implied universal quantifiers this enables any algebraic expression, such as

$$s_{i+1} = s_i + n_i, \quad (11)$$

to define infinitely many relations, called a class relation, and to be able to process an unlimited number of elements. The name part of the identifier identifies the class relation and the indices instantiate a particular relation to process that element.

The element-based approach does not use indices in the same way as the instruction-based model. An index is part of the element identifier and not an offset into some fixed memory. The element name and indices provide the semantic information required to process the element and thus this avoids having to use addressing to access data. The other advantage is that elements are never altered: they are created, processed and discarded; hence there are no cache memory writes. Again this facilitates distribution across processors.

The element model is computationally equivalent to the instruction model as it can implement a Turing machine. A series of examples, such as sorts and pattern matching, have been successfully implemented. Like with any model some algorithms are better suited to the element model. The purpose of this research is to explore how the element model is likely to perform for the specific case of simple matrix multiplication. The reason this case has been chosen is that it is a tightly coupled computationally intensive example.

III. Research Approach

What is the best way to evaluate the proposed model, especially only having von Neumann hardware available? The success of the model depends on it being able to compete head on with current technology. For this reason, the choice was made to do a direct comparison with the instruction-based model. Since the research is around finding better models for parallel computing, the case study needs to focus on this aspect. Experiments were run on a cluster as this was the only resource available. An MPI emulation was used for running the element based programs. A comparison is made between a MPI C program and an element-based program.

The model is evaluated in 3 ways: ease of programming; use of resources; and performance. The evaluation of the model is done on a number of levels. An important one is how easy it is to write programs. As the language has not been discussed in detail, a brief comparison is given. The next level is a complexity analysis of the resources required, looking at memory requirements and number of instructions executed and how this scales with the number of processors. Data collected in the emulation of the element-based model is used to verify the analysis. Finally a study is made of how the runtimes of the two programs compare as the size of the matrix and the number of processors increases.

At the end of the day what matters is whether the element-based model has the potential to out-perform the current technologies, and in particular, scale better with the number of processors.

A. The programs

To make a fair comparison and facilitate the analysis, the approach was to use the most basic design of the program for both cases. The algorithm for matrix multiplication of $\boldsymbol{A = B \times C}$ is simple:

$$\forall r \in [0, n_r - 1], c \in [0, n_c - 1], \quad A_{r,c} = \sum_{k=0}^{n_c-1} B_{r,k} * C_{k,c} \tag{12}$$

where n_c and n_r are the number of columns and rows. This can be expressed as a simple C program as follows:

```
for(row=0; row<NoRows;row++)
  for(col=0; col<NoCols;col++){
    A[row][col]=0;
    for(k=0;k<NoCols;k++)
        A[row][col] += B[row][k] * C[k][col]}
```

Figure 4: C Matrix Multiplication Code

In the case of the element base program one can almost use the expression as is. However to make the program more explicit we expand the above expression to:

$$\begin{array}{ll} \forall r \in [0, n_r - 1] & \\ \forall c \in [0, n_c - 1] & A_{r,c} = \sum_{k=0}^{n_c-1} B'_{r,k,c} * C'_{r,k,c} \\ & \\ \forall k \in [0, n_c - 1] & B'_{r,c,k} = B_{r,c} \\ \forall k \in [0, n_r - 1] & C'_{k,r,c} = C_{r,c} \end{array} \tag{13}$$

and further develop it to:

$$\begin{array}{l} B'_{r,c,k} = B_{r,c}\ k \in [1, n_c]; \quad C'_{k,r,c} = C_{r,c}\ k \in [1, n_r] \\ A^m_{r,k,c} = B'_{r,k,c} * C'_{r,k,c} \\ A^s_{r,n_c,c} = 0 \\ A^s_{r,k-1,c} = A^s_{r,k,c} + A^m_{r,k,c} \qquad \text{if } k \neq 0 \\ A_{r,c} = A^s_{r,0,c} \end{array} \tag{14}$$

The "assembly" code is given in Figure 4.

From	To	Op	Index *i*	Value *v*
b	b$0	xins	2	n-1
b$0	b$2	xnfltr	2	0
	a$0	tuple 0		
b$2	b$0	xsub	2	1
c	c$0	xins	0	n-1
c$0	c$2	xnfltr	0	0
	a$0	tuple 1		
c$2	c$0	xsub	0	1
a$0	a$1	mult		
a$1	a$2	xfltr	1	n-1
	a$3	xnfltr	1	n-1
a$2	a$4	xsub	1	1
a$3	a$5	tuple 0		
a$4	a$5	tuple 1		
a$5	a$6	add		
a$6	a$7	xfltr	1	0
	a$2	xnfltr	1	0
a$7	a	xdel	1	1
a	print	print		

Where

Op	Description
xins	inserts *v* in index position *i*
xsub	subtracts *v* from index$_i$
xfltr	if index$_i$ = *v* create new element
xnfltr	if index$_i$ ≠ *v* create new element
xdel	delete index in position *i*
tuple	form tuple
add	value$_0$ = value$_0$ + value$_1$
mult	value$_0$ = value$_0$ * value$_1$

Figure 4: Element based assembly code

The two programs appear similarly complex, but we now need to look at the parallel versions. In the case of the C program we chose to distribute the computation of each element of *A[r,c]* to a processor as outlined below.

```
// master
 for(row=0; row<NoRow;row++)
   for(col=0; col<NoCol;col++)
      send(nextproc,B[row],C[col]);
      receive(proc,A[row,col]);
// slave
 receive(master, B, C);
 result=0
 for(k=0; k<NoCol;k++)
       result+=B[k] * C[k];
 send(master, result);
```

Figure 5: MPI C Matrix Multiplication Code

The implementation of this is somewhat more complicated using MPI. Managing the slaves and synchronizing requires additional code provided in Figure 6.

The element-based program, by comparison, is inherently parallel and does not need to be rewritten to run on a parallel architecture. Also there are no built in restrictions as to the size of the buffers for message passing. Using an old measure of lines of code, the element-based program is a fraction of the size. There is no need to come to grips with message passing or synchronization. Not having to manage the parallelism gives the element-based approach an advantage.


```
int main(int argc, char *argv[])
{
  int slave, noslaves, result[4];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &slave);
  MPI_Comm_size(MPI_COMM_WORLD, &noslaves);
//=========================
// master slave
//=========================
if(slave==master_slave)
{
  int sizeM;
  scanf("%d",&sizeM);

  int A[sizeM][sizeM], B[sizeM][sizeM],
      C[sizeM][sizeM], row, col,result[5];
  MPI_Status status;

  //===========
  //  read in data
  //===========

  for(row=0; row<sizeM; row++)
    for(col=0; col<sizeM; col++)
    {
        MPI_Recv(result, 5, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG,MPI_COMM_WORLD,&status);
        if(result[1]!=-1)
                A[result[1]][result[2]]=result[3];
        result[1]=row;
        result[2]=col;
        result[3]=0;
        result[4]= sizeM;
        MPI_Send(&result, 5, MPI_INT, result[0],
                                resm,MPI_COMM_WORLD);
        MPI_Send(&B[row], sizeM, MPI_INT, result[0],
                                rowm,MPI_COMM_WORLD);
        MPI_Send(&C[col], sizeM, MPI_INT, result[0],
                                colm,MPI_COMM_WORLD);
  }
  for(row=0; row<noslaves-1; row++)
  {
    MPI_Recv(result, 5, MPI_INT, MPI_ANY_SOURCE,
                                resm,MPI_COMM_WORLD,&status);
    A[result[1]][result[2]]=result[3];
    result[1]=-1;
    MPI_Send(&result, 5, MPI_INT, result[0],
                                resm,MPI_COMM_WORLD);
  }
} //end Master
// =======
//  slave
//  =======
else
{

  int B[100], C[100], result[5], i;
  MPI_Status status;
  result[1]=-1;
  result[0]=slave;
  MPI_Send(&result, 5, MPI_INT, 0,
                                resm,MPI_COMM_WORLD);
  while(1)
  {
    MPI_Recv(result, 5, MPI_INT, 0,resm,
                                MPI_COMM_WORLD,&status);
    if(result[1]==-1)break;
    MPI_Recv(B, result[4], MPI_INT, 0,
                                rowm,MPI_COMM_WORLD,&status);
    MPI_Recv(C, result[4], MPI_INT, 0,
                                colm,MPI_COMM_WORLD, &status);
    result[3]=0;
    for(i=0;i<result[4];i++)
      result[3]+=B[i]*C[i];
    MPI_Send(&result, 5, MPI_INT, 0,
                                resm,MPI_COMM_WORLD);
  }
} //end slave
  MPI_Finalize();
}
```

Figure 6: Parallel MPI Matrix Multiplication Program

The assembly code generated by the parallel version of the C program ran to 600 odd lines and 20 lines for the slave. A small sample is given in Figure 7. Comparing the element-based code with the C code, there is a close mapping between the element program and the low level machine code, in contrast to the instruction-based. In the element-based model the semantic gap between the high-level program and the machine code is small resulting in a small amount of machine code. The second aspect is that the element-based program does not need to be altered to run on a parallel architecture. It is inherently parallel.

```
 1  movl   $100, -52(%rbp)      16  leaq 16(%rsp), %rcx
 2  movl   -52(%rbp), %edx      17  movq %rcx, -1192(%rbp)
 3  movslq %edx,%rax            18  movq -1192(%rbp), %rax
 4  salq   $2, %rax             19  addq $15, %rax
 5  movq   %rax, -1200(%rbp)    20  shrq $4, %rax
 6  movl   -52(%rbp), %eax      21  salq $4, %rax
 7  movslq %edx,%rdx            22  movq %rax, -1192(%rbp)
 8  cltq                        23  movq -1192(%rbp), %rax
 9  imulq  %rdx, %rax           24  movq %rax, -168(%rbp)
10  salq   $2, %rax             25  movl -52(%rbp), %edx
11  addq   $15, %rax            26  movl %edx, -1188(%rbp)
12  addq   $15, %rax            27  movslq -1188(%rbp),%rax
13  shrq   $4, %rax             28  salq $2, %rax
14  salq   $4, %rax             29  movq %rax, -1176(%rbp)
15  subq   %rax, %rsp           30  movl -52(%rbp), %eax
```

Figure 7: Sample of assembly code for MPI program

B. Complexity Analysis

The complexity of the instruction-based model can be measured in terms of the number of instructions, whereas for the element-based model, elements can be used. A starting point is to show that the complexity of the element-based model is at least equivalent, if not better, than the instruction-based model for the matrix example. Having established that the complexity of the element-based model is comparable with the instruction model, the focus can turn to a comparison between the execution times and scalability with increased number of processors.

The analysis is a waste of time if processing an element is clearly vastly more expensive than executing an instruction. On the basis of the instruction cycle versus the element cycle, creating an element is argued to be at least as efficient as processing an element. Consider each step in the cycle.

Accessing/storing elements from/to hardware queues should give the element-based approach a major advantage over having to read/write data from/to memory. The element-based approach has the potential of being able to delay paging to access relations until there is enough demand. Except for tuples, information is only read, eliminating the need for cache block write-back. Any number of elements can be processed in parallel, as there are no dependencies between them. This scaling factor has the potential to far exceed any pipelining approach of the instruction model. Even though it is not clear what the full cost of performing the tuple operation will be, this seems a reasonable initial assumption that the cost of creating an element is likely to be better than performing an instruction.

Table 8: Comparison of memory accesses

Step	Instruction-based	Element-based
1	get instruction	pop element
	memory access	stack register
	cache/page faults	none
2	get data	get relation
	memory/register	memory
	cache/page faults	cache/page faults
3	perform instruction	evaluate relation
	exception branch inst	exception tuple op
	cache/page fault	cache/page fault
4	store	push element
	memory/register	stack register
	cache/page faults	none

For the analysis, the multiplications of $n \times n$ matrices are compared. The master has an outer loop of 97 instructions and the inner loop is 92. The slave has 79 instructions of which 27 are in the loop. Every time the slave is passed a message it loops n times to compute one element in the resulting matrix resulting in $27n$ instructions. For every entity in the resulting matrix, a message is sent to the slave that is n^2 messages. Thus, the number of instructions computed in the slave's loop is $27n^3$. There are 52 instructions remaining in the slave if one excludes the instructions in the slave's loop and the 92 instructions in the inner loop of the master that are executed for each row and column. This gives 144 instructions that are executed n^2 times. There are 5 instructions in the outer master loop not in the inner loop that are executed n times and some constant c instructions that are only executed once in the master. Thus the number of instructions executed for a $n \times n$ matrix is $27n^3+144n^2+5n+c$. This ignores the cost of the calls to MPI subroutines that should result in a significant number of additional instructions that get executed.

Table 9 works out the number of elements processed to be $14n^3+7n^2$. It should be no surprise that the number of instruction/operations in both the instruction and element-based programs are of order n^3. Even so the number of elements processed, $14n^3+7n^2$, is more or less half the number of instructions executed by the instruction-based program, $27n^3+144n^2+5n+c$. However, the real issue is what improved performance can be expected by increasing the number of slaves and how the speed scales with increasing number of processors.

Consider the instruction-based C program first. Given m slaves, the calculation of n^2 entries of the resulting matrix is distributed to the m^3 slaves as these processors become available. Given an ideal environment, in which there are no delays in sending and receiving data, the speed up should be directly related to the number of processors. The purpose of the empirical study is to assess

- what the overhead of the MPI model is on achieving this as well as the cost of transferring the data and
- the weaknesses the instruction-based model has i.e. the memory wall, page faults etc.

The particular program that is used does have a limit on the size of the matrices it can handle. Other aspects that affect the performance are cache faults that can only be measured empirically.

Table 9: Analysis of Elements Processed

domain	range	op	number
	b		n^2
	c		n^2
b	b$0	xins	n^2
b$0	b$2	xnfltr	n^3
	a$0	tuple 0	n^3
b$2	b$0	xsub	n^3
c	c$0	xins	n^2
c$0	c$2	xins	n^3
	a$0	tuple 1	n^3
c$2	c$0	xsub	n^3
a$0	a$1	mult	n^3
a$1	a$2	xfltr	n^3
	a$3	xnfltr	n^3
a$2	a$4	xsub	n^3
a$3	a$5	tuple 0	n^3
a$4	a$5	tuple 1	n^3
a$5	a$6	add	n^3
a$6	a$7	xfltr	n^3
	a$2	xnfltr	n^3
a$7	a	zdel	n^2
a	print	print	n^2
			$14n^3+7n^2$

The same argument holds for the element-based model that the elements can be distributed across the m processors. There are however some advantages in terms of distributing the elements. Elements can be processed in any order and one does not need to synchronize the sending and receiving of elements between processors. The element-based model processes elements rather than accessing data via an address in memory, eliminating this potential cause of page faults. The element-based model does synchronize using a hash table that has potential page fault costs. Part of the empirical study is to get some indication of this cost. In doing the comparison of cost, the following factors need to be taken into account: the

[3] One processor is used for the master.

cost of emulation and the inability to tailor hardware to suit the element model. Processing an element involves executing 12 emulator C statements. The hardware used to run the comparison is highly optimized for the instruction-based model. The emulator has none of these benefits. Hardware queues and hardware designed to perform tuple hashing would improve the performance.

C. Empirical Study

In the previous section a case is made that, potentially, the element model has significant advantages over the instruction model. However the argument is based on a number of assumptions. This section tries to get some handle on how valid these assumptions are. The runtimes of the instruction-based and the element-based programs are collected for different sizes of matrices and different numbers of processors. These runtimes are compared to assess:

- whether the element-based program execution times are comparable with the instruction-based execution times, given that the code is emulated on an instruction-based computer, and
- whether the element-based model scales better with the number of processors.

An MPI program (see Figure 8) was used to emulate the behavior of the model to give some idea of its possible performance and highlight weaknesses. The results give some indication of the potential performance of an element-based architecture. An evaluation of the experiments is used to assess the validity of the assumptions made in the complexity analysis. On the basis of these results an assessment is made as to whether the element model is worth investigating further.

Experiments were undertaken to assess if there are grounds to support the arguments that an element-based model:

- can be implemented to do a computation such as matrix multiplication,
- performs better without parallelism, and
- scales better with increasing number of processors.

The experiments involve comparing the performance of the emulation of the element-based program and the execution of the C program. The following data is compared:

- the number of elements created with increasing sizes of matrices with a single slave,
- the wall times of both with increasing sizes of matrices with a single processor, and
- the wall times of both for a given size of matrix with increasing number of slaves.

The simple emulator shows that it is possible to implement the model of computation to compute problems such as matrix multiplication. The model does handle the essential primitives i.e. selection, iteration and progression[4].

The predicted number of elements relates closely to the number of elements processed. The difference with the analysis $14n^3+7n^2$ differs by a factor of n^2 shown by Table 10.

Table 10: Comparison Between Predicted and Actual

Size	Predicted	Actual
100	14 070 000,00	14 960 000,00
110	18 718 700,00	19 916 600,00
120	24 292 800,00	25 862 400,00
130	30 876 300,00	32 887 400,00
140	38 553 200,00	41 081 600,00
150	47 407 500,00	50 535 000,00

One reason for the discrepancy is an implementation design aspect that results in an additional element being created for each tuple operation. This is not a significant difference and aspects like system calls by the C program can result in similar discrepancies. This result supports the argument that the element-based model will perform better, given the assumption that creating an element is no worse than executing an instruction.

If we look at the execution times for the matrix of 100 by 100 in Table 11 we see that the number of instructions per time unit is 15.5 for the element based technique and 240.8 for the instruction based one. This is roughly 15.5 times slower than the C program. Considering that the emulator has to execute well over twenty instructions for processing each element, it suggests that the assumption, that processing an element is at least equivalent to processing an instruction, may be a reasonable one. If one takes into account that the estimated number of instructions executed is three times the number of elements evaluated, this reduces the gap to just over 5 times slower.

[4] The equivalent sequential execution in the instruction based form.

```
while(!stackEmpty)
{
  pop(element);
  new_element=element;
  nextRelation= relationTable[element.id].list;
  numberRelations = relationTable[element.id].size;
  for(i=0; i<numberRelations;i++)
  {
    new_element.id= relation[nextRelation].id;
    case(relation[nextRelation].op)
    {
      // arithmetic and logical operations
      add:
      {
        new_element.value[0] += element.value[1];
        break;
      }
      ...
      // index operations
      indexAdd:
      {
        new_element.index[relation[nextRelation].parm0]]
           += relation[nextRelation].parm1];
        break;
      }
      ...
      // if operation
      ifop:
      {
        if(!new_element.value[1])
          goto skip; // do not create a new element
        break;
      }
      ...

      joinL:
      {
        int found;
        checkHashTable(newElement, found);
        if(!found) goto skip;
        break;
      }
      ...
    }
    processSlave=
       slave_mapping(newElement.index,noslaves);
    push(processSlave,newElement);
    skip:
  }
}
```

Figure 8: Outline of Emulator

The picture does not look so favorable when the matrix size increases and the element-based model is from 5 times for (100 by 100) to 7.5 times slower for (150 by 150) see Table 11. However the emulation is at present a very simple implementation and on a totally unsuitable architecture. Like with the current architectures, considerable effort will need to be put into its design. Since elements do not need to be processed in a predetermined order, this enables the processor to better manage the resources by determining the order in which elements are processed. This could have significant impacts on the implementation of the tuple operations that could explain the deterioration of performance as the size of the matrices increase.

The next comparison given in Table 11 looks at how the two approaches speed up with the number of slaves. The instruction-based times do not improve beyond two slaves whereas we see that the times of the element-based program do. The execution times for the element-based program show some unexpected results, even though for all measures, the best wall time of five executions was selected. Part of the problem may be a property of the cluster. Both the instruction and element-based times seem out of line for the case with 6 slaves. The picture looks much better for the element-based program if we look at the speedup factor. The performance is better than linear up to 8 slaves and unexpectedly there is a degree of superscalar speedup. This result further supports the argument that this model needs to be explored further.

Table 11: Comparison of wall times

	Element-based		Instruction-based	
Size	Predicted	Time	Predicted	Time
100	14070	907	42141	175
110	18719	1211	56072	280
120	24293	1530	72778	322
130	30876	2052	92511	398
140	38553	2665	115523	410
150	47408	3299	142066	442

Table 12: Comparison of wall times

	Elem		Instr
Number Processors	Time	Speed up	Time
1	10207	1.0	978
2	3959	2.6	646
3	3315	3.1	627
4	2250	4.5	573
5	1728	5.9	672
6	1865	5.5	661
7	1193	8.6	620
8	1198	8.5	626

IV. Evaluation and Conclusion

This paper presents an alternative computational model that has significant advantages. The most significant is the concept of processing elements one at a time in a similar way to processing instructions. Underpinning this mechanism is that elements encapsulate both semantic information and data. The semantic information is used to determine how an element gets processed to create new elements. Like the von Neumann model, the evaluation cycle is a simple three-step process.

Since this paper is focused on the model as the basis for parallel architectures, the description of programs is touched on. The program in the case study is expressed in basic algebraic notation that maps closely to the machine level code. There is a small semantic gap between the program and the code, unlike the C program. Both the source and the assembly code are more concise than the equivalent C program. The element program has a big advantage in that it is inherently parallel and does not need to be specially designed for parallel processors. If the case study is representative of writing programs for parallel architectures, then the element base architecture has a decided advantage.

The complexity analysis of the two programs unsurprisingly shows that both are of order n^3. However, on the basis of the given assumptions and potential advantages of the element-based model, the analysis indicates that the element-based model is more efficient than instruction-based model. Both programs should exhibit linear speedup with the number of processors.

The element-based execution times were considerably better than was expected, given that the element-based execution used a crude emulator running on an instruction-based cluster. The area of concern is around the decline in performance as the matrix size increases. The suspected reason for this is the hashing with the associated page faults. More investigation is needed to explore how to improve the tuple hashing by ordering the processing of tuples. Being able to use hardware queues rather than addressed memory access for the unprocessed elements, as well as passing elements between processes, should further improve the performance.

The element-based execution times showed a better than linear speedup whereas the instruction-based did not. This contradicts the complexity analysis for the instruction base model that it would. This provides the strongest case that the element-based model may be a better model for parallel architectures.

This paper provides evidence that, for a tightly coupled memory intensive program, the element-based model has the potential to perform better than the instruction-based model. The next step is to implement the model using FPGAs [9]. However this is beyond my resources without some support for the potential of the model. Research is continuing informally, describing both the theory underpinning the model and the language.

REFERENCES

[1] John L. Hennessy David A. Patterson, "Computer Organization and Design: The Hardware/Software Interface", *Morgan Kaufmann*, 4th edition, (2009).

[2] Kreste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David P. Patterson, William Lester Plishker, John Shalf, Samuel Webb Wiliams, and Katherine A. Yelick, "The landscape of parallel computing research: A view from Berkeley", Technical report, *Electrical Engineering and Computer Sciences, University of California at Berkeley*, (2006).

[3] Klaus Erik Schauser David E. Culler and Thorsten von Eicken, "Two fundamental limits on dataflow multiprocessing", *Technical Report UCB/CSD-92-716, EECS Department, University of California, Berkeley*, (1992).

[4] David Patterson John Hennessy, "A conversation with John Hennessy and David Patterson", *ACM Queue*, 4(10), 2006.

[5] http://www.mpi forum.org/. Message passing interface forum.

[6] Arvind, "Data flow languages and architectures", ISCA '81: *Proceedings of the 8th annual symposium on Computer Architecture*, 1981.

[7] Sajjan G Shivas, *Advanced Computer Architecture*, pages 284-299, 2006.

[8] R.S.V. Whiting, P.G. Pascoe, "A history of data-flow languages", *Annals of the History of Computing, IEEE.*

[9] J. Teifel and R. Manohar, "An asynchronous dataflow fpg a architecture", *Computers*, IEEE Transactions on, 53(11):1376–1392, Nov. 2004.

Dr Conrad Mueller is a research fellow in both the School of Electrical and Information Engineering: University of the Witwatersrand and in the School of IT: Monash South Africa. His research interests are programming languages, computer architectures and computer science education with the main focus on new computational model that can contribute to these three fields. He received his PhD in Computer Science from The University of Witwatersrand. He had taught a wide range of courses in Computer Science. He is a member of the ACM and a fellow of the Institute of Computer Scientists and Information Technologists, South Africa.

