Effective Data Parallel Computation using the Psi Calculus

L. R. Mullin and M. A. Jenkins

CSC-94-02

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401

Effective Data Parallel Computation using the Psi Calculus *

Lenore M. R. Mullin Michael A. Jenkins †

January 24, 1994

1 Introduction

The essential difficulty in large scale parallel computation is to find an effective way to match the high level understanding of how a computational problem can be solved to the details of individual computational units and their organization into networks of processors on which the computation is to be carried out. Our primary interest in this paper is in large scale scientific and engineering computations which contain one or more steps requiring a massive computation that can utilize a parallel architecture to achieve significant speedup.

In current methodology, the problem will have normally been programmed (usually in Fortran) to be solved sequentially on a single processor, and the code is rewritten or automatically processed to run on a particular parallel architecture. The task of parallelizing sequential algorithms is very difficult, especially if the high level description has been lost in the effort to produce effective sequential code.

It is our view that effective utilization of parallel architectures can be achieved by using formal methods to describe both computations and computational organization within networks of processors. The idea is to return to the mathematical treatment of the problem as a high level numerical algorithm, describe it in an algorithmic formalism that captures the inherent parallelism of the computation, provide a high level description of the available architecture, and then use transformational techniques to convert the high level description into a program that utilizes the architecture effectively. The hope is that one formalism can be used to describe both computations and architectures and that a methodology for automatically transforming computations can be developed.

*This work was supported by the National Science Foundations's PFF Award Program and the Natural Sciences and Engineering Council of Canada

†L. Mullin, lenore@cs.umr.edu, Department of Computer Science, University of Missouri-Rolla, Rolla, Missouri 65401 USA and M. Jenkins, maj@qucis.queensu.ca, Department of Computing and Information Science, Queens University, Kingston, Ontario K7L 3N6 CANADA

The formalism and methodology described in the paper is a first step towards the ambitious goal described above. It uses a theory of array data structures as the formalism and two levels of conversions, one for simplification and another for data mapping.

Large scale scientific computations can be described in terms of computations on scalars, vectors, matrices and higher dimensional arrays which sometimes necessitate restructuring for compatible mapping to a multiprocessing architecture. The need for high level operations describing computations in such terms has been recognized by the scientific computation community leading to the effort to design a High Performance Fortran[8]. Many of the ideas have their origin in APL and other interpreted languages that support arrays, such as Nial[12] and Paralation Lisp[18]. There is considerable academic interest in finding a suitable formalism for parallel programming[19, 6]. Thus, the problem of developing an algorithmic formalism for describing scientific computations at a high level is being actively investigated and solutions are emerging. Some work has been done in developing transformation schemes for such formalisms but much of the work is aimed at sequential processing.

Similarly, much work has been done in describing actual architectures in terms of abstract models and then using the abstract model as the basis for mapping decisions[3, 4, 15]. However, it is still a primarily manual effort to design the mapping of the computation to the abstract model. The automation of this step is crucial if we are to make effective use of parallel architectures.

The abstract model for the organization of the processors is often in the form of a graph whereby the nodes of the graph are processors and the edges are communication links. For many practical models the graph can be represented by an array in which each processor is given an address and each processor to which a link is available is at an address one away along one of the axes. The array model describes a list of processors, a 2 dimensional mesh, a hypercube, or a balanced tree[3, 4].

Our approach will be to view an architecture as having two array organizations, one which is the abstract model best suited for describing the problem and a second which corresponds to an enumeration of the processors as a list. For an actual architecture the latter corresponds to the list of processor identity numbers and is used to determine the actual send and receive instructions issued by the resulting program.

In doing the mapping from the data arrays of the problem to the array-like arrangements of processors, there is a need to be able to systematically determine what information to distribute to which processors. Having made those decisions, the high level algorithm expressed in terms of array operations has to be turned into low level code that selects data elements from one-dimensional memory, sends it to the appropriate processor in the one-dimensional list of processors. Each processor has to be supplied with code that is parameterized so that it operates on the data in its local memory to carry out its portion of the parallel algorithm.

The difficulties are compounded when a problem is so large that it must be attacked in slices. Thus, a vector of length $k = m * n * p$, where m is the number of slices, n is the amount of data each processor can process in one go, and p is the number of processors, can be viewed algorithmically as a 3-dimensional array of shape m by n

by p, where the first axis indicates slices of work to be done one after the other.

The manipulations of the data addresses to ensure that the problem decompositions are handled correctly can be quite intricate and are difficult to get right by hand. Thus, having a formal technique for deriving the address computations, one that can be automated to a large extent, is essential if rapid progress is going to be made in exploiting parallel hardware for scientific computation.

The next section describes a formalism that we are using to describe array computations and describe array architectures. It is centered around a generalized array indexing operation that selects a partition of an array described by a partial index. All the other high level array operations are defined using the indexing operation. The central theorem indicates how to describe the effect of the indexing operation in terms of selecting items from the one-dimensional list of the items assumed to be in row major order. Section 3 describes the work under way to develop the transformation system to convert high level descriptions in the calculus to programs that use linear addressing of the memory and the list of processors.

2 The Psi Calculus

There have been several investigations into a mathematics to describe array computations. These include More's theory of arrays (AT)[13], a formal description of AT's nested arrays in a first order logic[11] and the development of a mathematics of arrays (MOA) [14]. In [9] the correspondence between AT and MOA is described.

The Psi Calculus is the core set of operations derived in MOA. Unlike other theories about arrays[17, 13], all operations in MOA are defined using shapes and the indexing function Psi(ψ). The Psi Calculus and MOA puts closure on concepts introduced by Abrams[1] to optimize the evaluation of APL array expressions. His work led to many attempts to compile APL, with varying degrees of success[5, 7].

MOA has focussed on describing computations on flat arrays of numbers. The central operation of MOA is the indexing function

$$p\psi A$$

in which a vector of n integers p is used to select an item of the n-dimensional array A. The operation is generalized to select a partition of A, so that if q has only k components then

$$q\psi A$$

is an array of dimensionality $n - k$ and q selects among the possible choices for the first k axes. In MOA zero origin indexing is assumed. For example, if A is the 3 by 5 by 4 array

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \end{bmatrix} \begin{bmatrix} 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 \\ 37 & 38 & 39 & 40 \end{bmatrix} \begin{bmatrix} 41 & 42 & 43 & 44 \\ 45 & 46 & 47 & 48 \\ 49 & 50 & 51 & 52 \\ 53 & 54 & 55 & 56 \\ 57 & 58 & 59 & 60 \end{bmatrix}$$

then

$$< 1 > \psi a = \begin{bmatrix} 21 & 22 & 22 & 23 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 \\ 37 & 38 & 39 & 40 \end{bmatrix}$$

$$< 2\ 1 > \psi a \ = \ < \ 45 \ \ 46 \ \ 47 \ \ 48 \ >$$

$$< 2\ 1\ 3 > \psi a \ = \ 48$$

From ψ and a few elementary vector operations the common array manipulation operations can be defined.

We now introduce notation to permit us to define ψ formally and to develop the *Psi Correspondence Theorem*, which is central to the effective exploitation of MOA in array computations. We will use $A, B, \ldots$ to denote an array of numbers (integers or reals). An array's dimensionality will be denoted by d_A and will be assumed to be n if not specified.

The shape of an array A, denoted by s_A, is a vector of integers of length d_A, each item giving the length of the corresponding axis. The number of items in an array, denoted by t_A, is equal to the product of the items of the shape. The subscripts will be omitted in contexts where the meaning is obvious.

An index is a vector of n integers that describes one position in an n-dimensional array. Each item of an index for A is less than the corresponding item of s_A. There are precisely t_A indices for an array. A partial index of A is a vector of $0 \leq k \leq n$ integers with each item less than the corresponding item of s_A.

We will use a tuple notation (omitting commas) to describe vectors of a fixed length. For example,

$$< i\ j\ k >$$

denotes a vector of length three. <> will denote the empty vector.

For every n-dimensional array A, there is a vector of the items of A, which we denote by the corresponding lower case letter, here a. The length of the vector of items is t_A. A vector is itself a one-dimensional array, whose shape is the one-item vector holding the length. Thus, for a, the vector of items of A, the shape of a is

$$s_a = < t_A >$$

and the number of items or tally of a is

$$t_a = t_A.$$

The precise mapping of A to a is determined by a one-to-one ordering function. Although the choice of ordering is arbitrary, it is essential in the following that a

specific one be assumed. By convention we assume the items of A are placed in a according to the lexicographic ordering of the indices of A. This is often referred to as *row major ordering*. Many programming languages lay out the items of multidimensional arrays in memory in a contiguous segment using this ordering. Unfortunately, Fortran uses the ordering corresponding to a transposed array in which the axes are reversed.

Scalars are introduced as arrays with an empty shape vector and a list of one item that contains the scalar. This self-nesting definition for scalars has been shown to be well founded in [11].

There are two equivalent ways of describing an array A:

1) by its shape and the vector of items, i.e.
$A = \{s_A, a\}$, or

2) by its shape and a function that defines the value at every index p.

These two forms have been shown to be formally equivalent in a first order theory of arrays[10]. We wish to use the second form in defining functions on multidimensional arrays using their cartesian coordinates (indices). The first form is used in describing address manipulations to achieve effective computation.

To complete our notational conventions, we assume that $p, q, ...$ will be used to denote indices or partial indices and that $u, v, ..$ will be used to denote arbitrary vectors of integers. In order to describe the i_{th} item of a vector a, either a_i or $a[i]$ will be used. If u is a vector of k integers all less than t_A, then $a[u]$ will denote the vector of length k, whose items are the items of a at positions u_j, $j = 1, ..., k-1$.

Before presenting the formal definition of the ψ indexing function we define a few functions on vectors:

u `cat` v	catentation of vectors u and v
$u + v$	itemwise vector addition assuming $t_u = t_v$
u * v	itemwise vector multiplication
$n + u$, $u + n$	addition of a scalar to each item of a vector
n * u, u * n	multiplication of each item of a vector by a scalar
$\iota\, n$	the vector of the first n integers starting from 0
$\pi\, v$	a scalar which is the product of the components of v
k `take` u	when $0 \leq k$ the vector of the first k items of u
	and when $k < 0$ the vector of the last k items of u
k `drop` u	when $0 \leq k$ the vector of $t_u - \|k\|$ last items of u
	and when $k < 0$ the vector of the first $t_u - k$ items of u
k `rotate` u	when $0 \leq k$ the vector of (k `drop` u) `cat` (k `take` u)
	and when $k < 0$ the vector or (k `take` u) `cat` (k `drop` u)

Definition 1 *Let A be an n-dimensional array and p a vector of integers. If p is an index of A.*

$$p\psi A = a[\gamma(s_A, p)],$$

where

$$\begin{aligned} \gamma(s_A, p) &= x_{n-1} \quad \textit{defined by the recurrence} \\ x_0 &= p_0, \\ x_j &= x_{j-1} * s_j + p_j, \quad j = 1, ..., n-1. \end{aligned}$$

If p is partial index of length $k < n$,

$$p\psi A = B$$

where the shape of B is

$$s_B = k \,\mathtt{drop}\, s_A,$$

and for every index q of B.

$$q\psi B = (p \,\mathtt{cat}\, q)\psi A$$

The definition uses the second form of specifying an array to define the result of a partial index. For the index case, the function $\gamma(s, p)$ is used to convert an index p to an integer giving the location of the corresponding item in the row major order list of items of an array of shape s. The recurrence computation for γ is the one used in most compilers for converting an index to a memory address[2].

¿From the definition it is easy to prove:

Corollary 1 $<> \psi\ A = A$.

and

Theorem 1 *Let A be an n-dimensional array and p a partial index so that $p = q \,\mathtt{cat}\, r$. Then*

$$p\psi A = r\psi(q\psi A).$$

Proof: The proof is a consequence of the fact that for vectors u, v, w

$$(u \,\mathtt{cat}\, v) \,\mathtt{cat}\, w = u \,\mathtt{cat}\, (v \,\mathtt{cat}\, w).$$

If we extend p to a full index by $p \,\mathtt{cat}\, p'$, then

$$\begin{aligned} p'\psi(p\psi A) &= (p \,\mathtt{cat}\, p')\psi A \\ &= ((q \,\mathtt{cat}\, r) \,\mathtt{cat}\, p')\psi A \\ &= (q \,\mathtt{cat}\, (r \,\mathtt{cat}\, p'))\psi A \\ &= (r \,\mathtt{cat}\, p')\psi(q\psi A) \\ &= r\psi(q\psi A) \end{aligned}$$

Theorem 2 (Psi Correspondence Theorem) *Let A be an n-dimensional array and p a partial index for A of length k satisfying $0 \leq k \leq n$. Then*

$$p\psi A = \{k \,\mathtt{drop}\, s_A, a[\,\mathtt{start}\,(s_A, p) * \,\mathtt{stride}\,(s_A, p) + \iota\pi(k \,\mathtt{drop}\, s_A)]\}$$

where

$$\mathtt{start}\,(s, p) = \gamma(k \,\mathtt{take}\, s, p),$$

and

$$\mathtt{stride}\,(s, p) = \pi(k \,\mathtt{drop}\, s).$$

Proof: By the definition of ψ

$$\begin{aligned} p\psi A &= \{s_B, b\} \\ &= \{k \,\mathtt{drop}\, s_A, b[\iota t_B]\} \\ &= \{k \,\mathtt{drop}\, s_A, b[\iota\pi(k \,\mathtt{drop}\, s_A)]\} \end{aligned}$$

If we compare this with the theorem statement we see that to complete the proof we have to show that

$$b[i] = a[\,\mathtt{start}\,(s, p) * \,\mathtt{stride}\,(s, p) + i]$$

for $0 \leq i < \pi(k \,\mathtt{drop}\, s_A)$.

For a given i, let q represent the corresponding index of B. Then

$$\begin{aligned} b[i] &= q\psi B \\ &= (p \,\mathtt{cat}\, q)\psi A \quad \textit{by the definition of } \psi \\ &= a[\gamma(p \,\mathtt{cat}\, q, s_A)] \end{aligned}$$

and the proof reduces to showing that for $0 \leq i < \pi(k \,\mathtt{drop}\, A)$,

$$\gamma(p \,\mathtt{cat}\, q, s_A) = \,\mathtt{start}\,(s, p) * \,\mathtt{stride}\,(s, p) + \gamma(q, k \,\mathtt{drop}\, s_A). \tag{1}$$

On the left of (1)

$$\gamma(p \,\mathtt{cat}\, q, s_A) = x_{n-1}$$

where

$$\begin{aligned} x_0 &= p_0 \\ x_j &= x_{j-1} * s_j + p_j, \quad j = 1, ..., k-1 \\ x_j &= x_{j-1} * s_j + q_{j-k} \quad j = k, ..., n-1. \end{aligned}$$

Thus

$$x_{n-1} = (...(x_{k-1} * s_k + q_0) * s_{k+1}...) * s_{n-1}) + q_{n-k-1}. \tag{2}$$

$$\begin{aligned} \mathtt{start}\,(s, p) &= \gamma(k \,\mathtt{take}\, s, p) \\ &= x_{k-1}. \end{aligned}$$

$$\begin{aligned} \texttt{stride}\,(s,p) &= s_k * \ldots * s_{n-1} \\ \gamma(q, k\,\texttt{drop}\,s_A) &= y_{n-1}. \end{aligned}$$

where

$$\begin{aligned} y_0 &= q_0 \\ y_j &= y_{j-1} * s_{j+k} + q_j. \end{aligned}$$

Thus, the right hand side of (1) becomes

$$\begin{aligned} \texttt{start}\,(s,p) * \texttt{stride}\,(s,p) + \gamma(q, k\,\texttt{drop}\,s_A) &= x_{k-1} * s_k * \ldots s_{n-1} + y_{n-1} \\ &= x_{k-1} * s_k * \ldots s_{n-1} + (\ldots(q_0 * s_k + q_1) * s_{k+1} \ldots) * s_{n-1} + q_{n-k+1} \end{aligned}$$

which is a partially factored version of (2) and the proof is complete.

The practical significance of the the Psi Correspondence Theorem is that because all the operations in the Psi Calculus are defined in terms of ψ their definitions can be translated from one involving cartesian coordinates into one involving selection from the list of items stored in memory or through abstract data or processor restructurings, their lexicographic location(s).

For example, consider definitions of `take` and `reverse` for multidimensional arrays.

Definition 2 (take) *Let A be an n-dimensional array, and k a non-negative integer such that $0 \leq k < s_0$. Then*

$$k\,\texttt{take}\,A = B$$

where

$$s_B =< k > \texttt{cat}\,(1\,\texttt{drop}\,s_A)$$

and for every index p of B,

$$p\psi B = p\psi A.$$

(In MOA `take` is also defined for negative integers and is generalized to any vector u with its absolute value vector a partial index of A. The details are omitted here.)

Definition 3 (reverse) *Let A be an n-dimensional array. Then*

$$s_{\texttt{reverse}\,A} = s_A$$

and for every integer i, $0 \leq i < s_0$,

$$< i > \psi\,\texttt{reverse}\,A =< s_0 - i - 1 > \psi A.$$

This definition of `reverse` does a reversal of the 0th axis of A.

Now consider the evaluation of the expression

$$< 1 \;\; 2 > \psi(2\,\texttt{take reverse}\,A) \tag{3}$$

where A is the array given in the previous section. The shape of the result is

$$\begin{aligned} & 2\,\mathtt{drop}\,s_{2\,\mathtt{take}\,\mathtt{reverse}\,A} \\ = \;& 2\,\mathtt{drop}\,(<2>\;\mathtt{cat}\,(1\,\mathtt{drop}\,s_{\mathtt{reverse}\,A})) \\ = \;& 2\,\mathtt{drop}\,(<2>\;\mathtt{cat}\,(1\,\mathtt{drop}\,s_A)) \\ = \;& 2\,\mathtt{drop}\,(<2>\;\mathtt{cat}\;<5\;4>) \\ = \;& 2\,\mathtt{drop}\;<3\;5\;4> \\ = \;& <4>\,. \end{aligned}$$

The expression can be simplified using the definitions:

$$\begin{aligned} & <1\;3>\psi(2\,\mathtt{take}\;\mathtt{reverse}\,A) \\ = \;& <1\;3>\psi\,\mathtt{reverse}\,A \\ = \;& <3>\psi(<1>\psi\,\mathtt{reverse}\,A) \\ = \;& <3>\psi(<3-2>\psi A) \\ = \;& <1\;3>\psi A \end{aligned} \tag{4}$$

This process of simplifying the expression for the item in terms of its cartesian coordinates is called *Psi Reduction*. The operations of MOA have been designed so that all expressions can be reduced to a minimal normal form [16].

We can now use the Psi Correspondence Theorem to find the algorithm to compute the expression directly.

$$\begin{aligned} & <1\;3>\psi(2\,\mathtt{take}\;\mathtt{reverse}\,A) \\ = \;& <1\;3>\psi A \\ = \;& \{<4>, a[\,\mathtt{start}\,(<3\;5\;4>,<1\;3>) * \mathtt{stride}\,(<3\;5\;4>,<1\;3>) \\ & +\iota\pi(2\,\mathtt{drop}\;<3\;5\;4>)]\} \\ = \;& \{<4>, a[8*4+\iota 4]\} \\ = \;& \{<4>, a[<32\;33\;34\;35>]\} \\ = \;& <33\;34\;35\;36> \end{aligned} \tag{5}$$

Thus, we have seen that an array expression involving operations of the Psi Calculus can be reduced to an expression only involving the operation ψ using Psi Reduction and then the Psi Correspondence Theorem is used to express the selection in terms of starts, strides and lengths. The latter can then be used to achieve memory access efficiently. A project is underway to construct a compiler that can take a high level problem description in the Psi Calculus and mechanically reduce its normal form targeting the C programming language. Figure 1 illustrates the input to the compiler in a linearized MOA notation. Figure 2 shows the output of the compiler as a C program that executes the above example[1]

```
main()
{
  #This is a program representing expression (3) which
  #is input to the Psi Compiler based on the grammar
  #for the Psi Calculus

  array R^1 <4>;
  const array A^3 <3 5 4> = < 1 2 3 4 5 6 7 8 9
                              10 11 12 13 14 15 16 17 18 19
                              20 21 22 23 24 25 26 27 28 29
                              30 31 32 33 34 35 36 37 37 39
                              40 41 42 43 44 45 46 47 48 49
                              50 51 52 53 54 55 56 57 58 59 60 >;
  R=<1 2> psi (<3> ptake
      (( (<5> - <1>) - omega <1 1> ( iota <5>)) psi omega <1 1> A);

}
```

Figure 1: Psi Compiler Input

The execution of the C program in Figure 2 augmented by a print statement would produce the following output:

```
33.000000 34.000000 35.000000 36.000000
```

which is the same result as the hand derivaton in (4).

This example shows that a simple high level description of a computation in the Psi Calculus can be mechanically translated into a C program that does the computation. Examination of Figure 2 reveals that the generated program contains redundant code, some of which is eliminated by an optimizing C compiler. Work is underway to improve the code produced by the compiler.

3 Mapping Tasks to Processors using the Psi Calculus

We can also use the idea of mapping cartesian coordinates to their lexicographic ordering when we want to partition and map arrays to a multiprocessor topology in a portable, scalable way. Consider, for example, a parallel vector-matrix multiply of vector A and matrix B, where $s_A =< n >$ and $s_B =< np >$. One effective way to

[1] Note that reverse is defined using the higher order operator omega[14] which partitions A into one dimensional vectors and indexes each vector from the rear to the front to yield a reverse.

```
main()
{
  #This would be the output of the Psi Compiler which mechanically
  #performed Psi Reduction thus reducing it to the same normal form
  #as expression(5).
  int i0; int i1; int i2;
  int step1[3]; int step2[3]; int shift,offset;
  float _R[4];
  float _A[]={0.0,  1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0,  9.0,
              10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0,
              20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0,
              30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0, 37.0, 39.0,
              40.0, 41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0, 48.0, 49.0,
              50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0, 57.0, 58.0, 59.0};
  int forall_i0[1]; int forall_i1[1];
  float *tmp_vect0; float consta0[1]; float consta1[1];
  for (forall_i1[0]=1; forall_i1[0]<1+1; forall_i1[0]++) {
      tmp_vect0=(float *) malloc((1)*sizeof(float));
    for (forall_i0[0]=(forall_i1[0+0]);
forall_i0[0]<(forall_i1[0+0])+1; forall_i0[0]++) {
      consta0[0]=5.000000;
      shift=(forall_i0[0+0]-forall_i1[0+0]);
      offset=0;
      *(tmp_vect0+shift)=*(consta0+offset);
      consta1[0]=1.000000;
      shift=(forall_i0[0+0]-forall_i1[0+0]);
      offset=0;
      *(tmp_vect0+shift)-=*(consta1+offset);

      shift=(forall_i0[0+0]-forall_i1[0+0]);
      *(tmp_vect0+shift)-=(forall_i0[0+0]);
      shift++;
    }
    shift=(forall_i1[0+0]-1);
    offset=(tmp_vect0[0+0])*12+(2)*4+(0);
    for (i2=0; i2<4; i2++) {
      *(_R+shift)=*(_A+offset);
      shift++;
      offset++;
    }
  }
}
```

Figure 2: Psi Compiler Output

organize the computation is to map each of the rows of B to a processor, send the elements of A to the corresponding processors, do an integer-vector multiply to form vectors in each processor, and then add the vectors pointwise to produce the result. The last step involves the adding together of n vectors and is best done by adding pairs of vectors in parallel. Abstractly, this last step can be seen as adding together the rows of a matrix pointwise. A hypercube topology is ideal for such a computation and at best would take $\mathcal{O}(\log n)$ on n processors to compute.

Often a hypercube topology is not available, we may only have a LAN of workstations or a linear list of processors. But, we can view any processor topology abstractly as a hypercube and map the rows to processors by imposing an ordering on the p available processors. That is we look at p_i where $0 \leq i < p$ as the lexicographically ordered items of the hypercube.

Hence, in the case of the LAN we obtain a vector of socket addresses. We then abstractly restructure the vector of addresses as a k-dimensional hypercube where $k = \lceil \log_2 n \rceil$. In order to map the matrix to the abstract hypercube we restructure the matrix into a 3-dimensional array such that there is a 1-1 correspondence between the restructured array's planes and the available processors. That is, we send the i^{th} plane of the restructured array to the i^{th} processor lexicographically. If there are more rows than processors then the planes are sequentially reduced within each processor in parallel.

We can apply the Psi Correspondence Theorem to the data to see how to address the i^{th} planes from memory efficiently. The same methodology can be applied to address the processors effectively.

For example, suppose we want to add up the rows of a 256 by 512 matrix and we have 8 workstations connected by a LAN. We would restructure the matrix into a 8 by 32 by 512 array which we denote by A'. The socket address of the workstations are put into a matrix P and each $< i > \psi A', i = 1, ..., 8$ is sent to the processor addressed by P_i. The sum of the rows for each plane are formed in parallel producing 8 vectors of length 512 in each processor.

We then restructure P into a 3-d hypercube implicitly and use this arrangement to decide how to perform the access and subsequent addition between the processors. In the first step we add processor plane 1 to plane 0. By the Psi Correspondence Theorem this implies adding the contents of processors 4 to 7 to those of processors 0 to 3. In the next step we add processor row 1 to row 0, which implies the contents of processors 2 and 3 are added to those of processors 0 and 1. Finally, we add the contents of processor 1 to the contents of processor 0. Thus, we have added up all the rows in $\log_2 8$ or 3 steps.

The method can be employed for any size matrix and can utilize an arbitrary number of homogeneous workstations connected by a LAN. It is a portable scalable design. A more detailed description of this technique (including timing results) can be found in [15]. Recently, we ported and scaled our designs to a 32 processor CM5. Presently, we are scaling our workstation design to a network of 16 SUN stations. We will run our experiments again on 16 processors of the CM5 and compare the performance of the two environments. We plan to add the knowledge to do mappings

of this sort to the compiler described in the previous section.

4 Conclusion

It is recognized that many large scale scientific and engineering problems have massive computational requirements that can utilize the collective power of large numbers of processors working together. The difficulty facing the computer science community is to provide tools to the scientists and engineers that allow them to solve such problems on large networks of workstations or specific parallel architectures in an effective and timely manner.

The approach we are espousing is that a formal methodology be developed that assists in automating the translation of high level descriptions of computationally intensive problems working from a high level array-based description of the problem. The paper discusses a specific formalism, the Psi Calculus, and demonstrates how by a combination of using Psi Reduction and the Psi Correspondence Theorem, progress has been made on both the problem of expressing problems at a high level, and on using the formalism both to generate low level code, and to assist in the organization of the computation on a processor network.

Our goal is to prototype tools that can be adopted by compiler designers to assist in the problem of parallelizing programs. We see this happening in two directions:

- by a language providing a high level notation embedded within a standard language, that is preprocessed using the techniques described in the paper, or
- by the compiler extracting a high level description based on loop analysis and then using the techniques to achieve an effective translation.

The next step towards our goal is to apply the methodology to a problem of practical size and complexity to demonstrate that significant scientific and engineering computations can be solved effectively in this manner.

References

[1] P.S. Abrams. *An APL Machine.* PhD thesis, Stanford University, 1970.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, Reading, Mass, 1986.

[3] N. Bélanger, L. Mullin, and Y. Savaria. Formal methods for the partitioning, scheduling and routing of arrays on a hierarchical bus multiprocessing architecture. Technical Report CSEE/92/06-04, University of Vermont, Dept of CSEE, 1992. Presented at ATABLE92, Montréal Québec, 1992, to be in proceedings.

[4] N. Bélanger, Y. Savaria, and L. Mullin. Data structure and algorithms for partitioning arrays on a multiprocessor. Technical report, University of Missouri-Rolla, Dept of CS, 1994. in progress.

[5] R. Bernecky. Compiling apl. *Arrays, Functional Languages and Parallel Systems*, 1991.

[6] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[7] T. Budd. A parallel intermediate representation based on lambda expressions. *Arrays, Functional Languages and Parallel Systems*, 1991.

[8] High Performance Fortran Forum. *High Performance Fortran Language Specification*, version 1 edition, May 1993.

[9] M. Jenkins and L. Mullin. A comparrison of array theory and a mathematics of arrays. In *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 1991.

[10] M. A. Jenkins. Arrays and functional programming. in progress, 1994.

[11] M. A. Jenkins and J. I. Glasgow. A logical basis for nested array data structures. *Computer Languages*, 14(1):35–51, 1989.

[12] M. A. Jenkins, J. I. Glasgow, Carl McCrosky, and H. Meijer. Expressing parallel algorithms in nial. *Journal of Parallel Computing*, 11(3), 1989.

[13] T. More. Axioms and theorems for arrays. *IBM Journal of Research and Development*, 17(2), 1973.

[14] L. M. R. Mullin. *A Mathematics of Arrays*. Ph.D. dissertation, Syracuse University, December 1988.

[15] L.R. Mullin, D. Dooling, E. Sandberg, and S. Thibault. Formal methods for scheduling and communication protocol. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, july 1993.

[16] L.R. Mullin, E. M. Insall, and W. Kluge. The psi calculus and the church-rosser property. in progress, 1994.

[17] E.M. Palvast. *Programming for Parallelism and Compiling for Efficiency*. PhD thesis, Delft University of Technology, June 1992.

[18] G. W. Sabot. *The PAralation Model: Architecture-Independent Parallel programming*. MIT Press, Cambridge, Mass., 1988.

[19] D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, December 1990.