[9] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a mechanism for intergrated communication and computation. In *Proceedings 19th Annual International Symposium on Computer Architecture*, pages 256–266. ACM Press, 1992.

[10] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

cost of implementing demand-driven eduction in the generator is negligible.

Clearly, it is important to not only exercise the GLU programming model in expressing diverse applications, but also to develop abstract architectures that are inherently more scalable, thus effectively exploiting less coarse-grain parallelism.

References

[1] E.A. Ashcroft. Dataflow and Eduction: Data-driven and demand-driven distributed computation. In *Current Trends in Concurrency*. Springer Verlag, 1986. Lecture Notes in Computer Science (224).

[2] E.A. Ashcroft. Tournament computations. In *Third International Symposium on Lucid and Intensional Programming*, Queen's University, Kingston, Ontario, Canada, 1990.

[3] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional Declarative Programming*. Oxford University Press, 1994. (In Preparation).

[4] N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, April 1989.

[5] A.A. Faustini and R. Jagannathan. Multidimensional programming in Lucid. Technical Report SRI-CSL-93-03, Computer Science Laboratory, SRI International, Menlo Park, California 94025, USA, January 1993.

[6] R. Jagannathan. A descriptive and prescriptive model for dataflow semantics. Technical Report CSL-88-5, Computer Science Laboratory, SRI International, Menlo Park, California 94025, May 1988.

[7] R. Jagannathan and A.A. Faustini. Tournament computations in GLU. In *Fourth International Symposium on Lucid and Intensional Programming*, Computer Science Laboratory, SRI International, Menlo Park, California 94025, 1991.

[8] V. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

cessors with relatively medium-grain parallelism, sustaining this efficiency on a large workstation cluster is only possible if remote computing time is at least two orders of magnitude larger than the associated communication time.

This study also confirms the fairly obvious intuition that the generator is the bottleneck of the generator/executor abstract architecture. There are two aspects to the generator bottleneck: one is the cost of communicating with all the executors and the other is the time spent in local function execution (i.e., the serial part of the application). And importantly, the generator spends only nominal amount of its time on eductive evaluation of the application kernels that were studied.

6 Concluding Remarks

The main contribution of this chapter is the validation of a dataflow-based approach to programming conventional parallel computers by showing that standard parallel application kernels can be succinctly expressed while reusing existing sequential code and that these kernels exhibit parallel performance which are comparable to those of equivalent kernels developed in lower-level explicit parallel programming systems.

In particular, we have described a coarse-grain dataflow system for programming conventional parallel computers. The system is based on a hybrid model of programming (GLU) that consists of a multidimensional dataflow language Lucid for composing implicitly parallel programs using imperatively-specified functions that specify computations.

We have shown how GLU can be used to succinctly compose applications with substantial inherent parallelism from existing sequential code with only nominal modifications.

We have shown how GLU programs can be mapped to abstract architectures which then can be compiled to specific target computers. Using the generator/executor abstract architecture, we have considered the efficiency of GLU programs executing in parallel on workstation networks. The speedup efficiency observed (speedup being relative to sequential execution on one processor) suggests that the generator eventually becomes the bottleneck mainly because communication cost and local function execution cost begin to dominate. We also observe that coarser granularity of parallelism means that better scaling of performance. Importantly, the performance study shows that GLU itself is not the source of any degradation since the

Processors	Granularity	Generator Utilization			Executor
		Communication	Eduction	Local Function	Utilization
1	127	0.8	0.1	1.2	98.2
2	182	1.1	0.1	1.8	97.8
4	128	2.9	0.3	3.8	92.5
8	168	4.1	0.4	6.1	86.7
16	150	8.2	2.0	11.5	77.2

Table 2: Prime Number Generation Performance

.

than mergesort shows degradation in performance as the number of processors increase. Like with mergesort, this is because the generator being busy results in the executors to idle between function applications as evidenced by lower average executor utilization with higher number of processors.

Table 3 shows the granularity and the utilizations of the generator and executors with varying number of processors for the matrix multiplication kernel. Unlike mergesort and prime number generation, the granularity

Processors	Granularity	Generator Utilization			Executor
		Communication	Eduction	Local Function	Utilization
1	17	5.5	0.1	3.9	91.5
2	18	10.0	0.2	6.8	88.9
4	18	18.7	0.2	12.8	84.7
8	18	32.6	0.3	22.6	74.1
16	18	53.4	1.3	34.3	59.8

Table 3: Matrix Multiplication Performance

.

of parallelism is much lower. This causes the generator to be busier with both communication and local function execution resulting in lower average executor utilization. Thus, the degradation in speedup efficiency is more pronounced as the number of processors increase.

5.3 Results Discussion

This performance study illustrates that coarser granularity of parallelism implies that performance can be scaled with increasing number of processors. While very good speedup efficiency can be achieved for small number of pro-

real matrices decomposed into 200 by 200 submatrices with only `mult` being executed remotely.

The performance of all three application kernels is above 85% for upto four processors. While mergesort and prime number generation degrade to slightly over 70% with 16 processors, matrix multiplication shows a speedup efficiency of 60% with 16 processors.

5.2 Detailed Performance

Let us consider the performance of each of the application kernels in terms of granularity of parallelism which is defined as the ratio of executor compute time to executor communication time, generator utilization which is the sum of the percentages of the total elapsed time used by the generator in communicating with executors, in eductive evaluation, and in executing "local" imperative functions, and executor utilization which is the fraction of the total elapsed time during which the executor is busy computing.

Table 1 shows the granularity, generator utilization, and executor utilization for the mergesort application kernel. The principal reason that

Processors	Granularity	Generator Utilization			Executor
		Communication	Eduction	Local Function	Utilization
1	84	1.2	0.0	1.2	98.1
2	102	1.9	0.0	2.8	94.9
4	93	4.0	0.6	4.3	91.8
8	91	7.9	0.1	8.7	89.3
16	93	13.2	0.1	15.2	77.1

Table 1: Mergesort Performance

.

the performance of mergesort degrades with increased number of processors despite having a high granularity of parallelism is because of increased generator utilization. Of the three components of generator utilization, only the overhead associated with eductive evaluation stays negligible. Both communication overhead and local function utilization increase with the number of processors. And correspondingly the average executor utilization decreases.

Table 2 shows the granularity and the utilizations of the generator and executors with varying number of processors for the prime number generation kernel. This application kernel despite having even higher granularity

tional parallel computer, we use a network of workstations on a local area network. Each workstation is a Sun SparcStation 2 with at least 16 megabytes of internal memory, sufficient swap space, and very little other user activity. The local area network is a 10 megabits per second ethernet.

5.1 Application Kernels Speedup

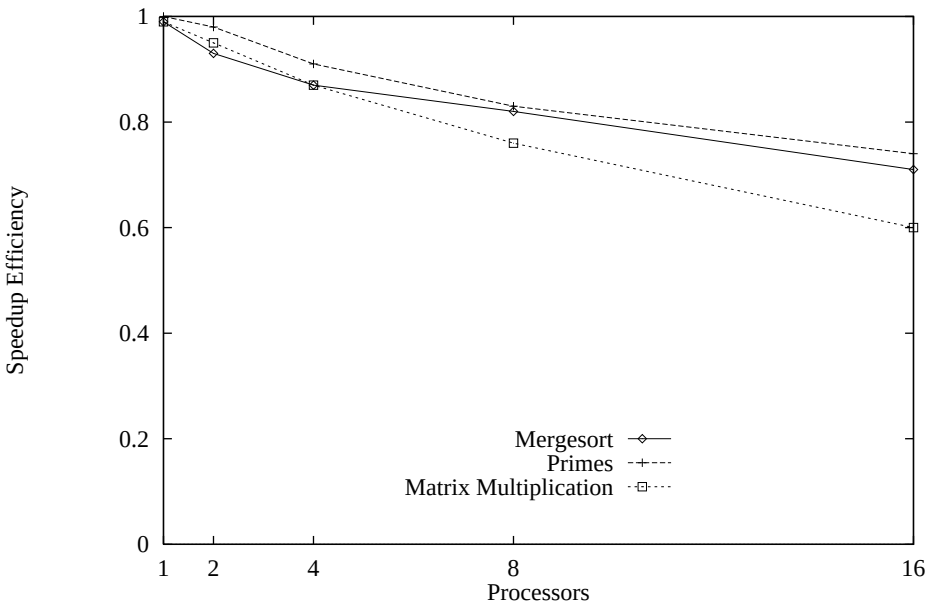

Figure 2: Performance of GLU Application Kernels on Workstation Cluster

Figure 2 shows the speedup efficiencies of the three application kernels, mergesort, prime number generation, and matrix multiplication. The mergesort kernel sorts 2^{20} numbers using 16 initial subsequences with only the initial `sort` being executed remotely. The prime number generation kernel finds all primes no greater than 4 million with only `seq_gen` being executed remotely. And the matrix multiplication kernel multiplies two 800 by 800

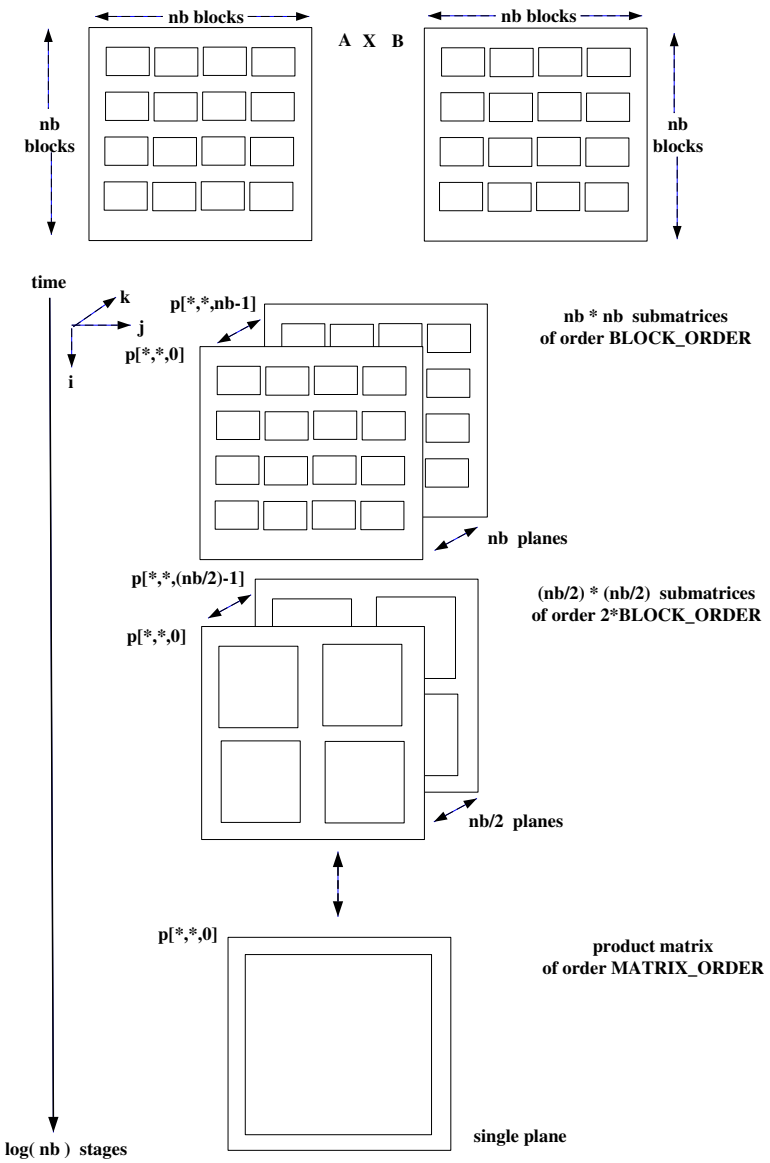

Figure 1: Block matrix multiplication using `cubic_tree` template

the inner product of rows of submatrices of `A` and columns of submatrices of `B` are computed. This results in a vector of planes where the K^{th} plane denotes the submatrices obtained by multiplying the K^{th} submatrix of each row of `A` with the K^{th} submatrix of each column of `B`. Another way of viewing this is as follows: the vector denoted by the $(I, J)^{th}$ submatrix of each plane corresponds to the product of the I^{th} row of submatrices of `A` with the J^{th} column of submatrices of `B`. At each subsequent stage, identically-positioned submatrices of adjacent planes are added pointwise thus reducing each adjacent pair of planes of submatrices to a single plane of submatrices. And, quartets of adjacent submatrices are juxtaposed into a single submatrix thus reducing the number of submatrices per plane by four with each resulting submatrix having four times as many elements. Eventually, after $\lceil \log_2(\texttt{nb}) \rceil$ stages, this results in a single plane consisting of a single matrix of the same order as the multiplicand matrices `A` and `B`. This corresponds to the product of the two matrices.

In the application kernels that we have considered, we have seen repeated expression of tournament parallelism using `linear_tree` and `cubic_tree`. This is because tournament parallelism is inherent in several otherwise diverse applications [7].

5 Parallel Execution of GLU Programs

The principal objective of this section on performance is to show that GLU programs despite being expressed at a very high-level of abstraction do not pay for this by being inefficient on conventional parallel computers. We show this by using a particular abstract architecture, namely the generator/executor architecture, which is inherently limited in the extent to which it can scale.

The most appropriate measure of performance of parallel execution of GLU programs is speedup efficiency. It is defined as the ratio of time taken by a sequential version of a program on a single processor divided by the product of the time taken of the GLU version on multiple processors and the number of processors. When speedup efficiency to be close to 100%, program parallelism is being efficiently exploited. And when speedup efficiency is sustained with increased number of processors, it means that the performance of the program is scalable.

We consider the three programs that we have described thus far: mergesort, prime number generation, and matrix multiplication. For a conven-

```
// matrix multiplication of two matrices of order MATRIX_ORDER
// using blocks of order BLOCK_ORDER

matmult( A, B )
where
  A = in( 0, MATRIX_ORDER );
  B = in( 1, MATRIX_ORDER );
  nb = ( MATRIX_ORDER / BLOCK_ORDER );
  matmult( A, B ) = cubic_tree.i,j,k( juxtapose, p, nb^3 )
    where
      dimension i, j, k;
        p = mult( a @.j k, b @.i k );
        juxtapose( m000, m001, m010, m011, m100, m101, m110, m111 ) =
                   combine( add( m000, m001 ), add( m010, m011 ),
                            add( m100, m101 ), add( m110, m111 )
                          );
        a = extract( A, i, j, BLOCK_ORDER );
        b = extract( B, i, j, BLOCK_ORDER );
    end;
end
```

The external imperative functions `mult`, `combine`, `add`, `extract` and `in` can be constructed with only nominal stylistic changes to existing sequential code. Variables `A` and `B` denote the names of the two matrices to be multiplied. Function `extract` selects the `(i,j)th` block (of size `BLOCK_ORDER`) from the argument matrix and returns it as the result. Variables `a` and `b`, which vary in both the `i` and `j` dimension, denote the extracted submatrices.

Figure 1 illustrates the computation specified by the above GLU program. It should be evident that at the initial stage, there is data parallelism in the simultaneous multiplications of each submatrix of `A` with each submatrix of `B`. In particular, assuming $\texttt{nb}^2$ submatrices per multiplicand matrix, $\texttt{nb}^3$ simultaneous submatrix multiplications can occur in parallel. The granularity of parallelism is relatively coarse since it corresponds to $\texttt{BLOCK_ORDER}^3$ multiplications. At each subsequent stage, there is geometrically decreasing data parallelism in the simultaneous addition and juxtaposition of submatrices which are in geometrically increasing order.

Let us consider the parallelism inherent in the program. The program is defined in terms of the `cubic_tree` template which corresponds to a 3-dimensional tree computation. At the first stage of the tree computation,

the product of A and B in terms of these $n/2$ by $n/2$ matrices.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$C_{11} = A_{11}B_{11} + A_{12}B_{21},$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22},$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21},$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Observe that the "divide-and-conquer" approach can be applied recursively to multiplication of the smaller matrices. Further observe that the computation of each of the eight products of the $n/2$ by $n/2$ matrices can occur simultaneously.

We present the GLU program for the solution which completely avoids division through recursion (and the associated overhead cost). Instead, it simply builds the product matrix by repeated addition and juxtaposition of products of submatrices.

The program is as follows:

```
// external data structure definition
struct matrix
   {
        double v[x][y];
   };
typedef struct matrix *MATRIX;

// external imperative function prototype definitions
MATRIX mult( MATRIX, MATRIX );
MATRIX juxtapose( MATRIX, MATRIX, MATRIX, MATRIX,
                  MATRIX, MATRIX, MATRIX, MATRIX
                );
MATRIX combine( MATRIX, MATRIX, MATRIX, MATRIX );
MATRIX add( MATRIX, MATRIX );
MATRIX extract( MATRIX, int, int, int );
MATRIX in( int, int );

// top-level Lucid composition of
```

The program uses four imperative functions: `initial_primes`, `catenate`, `rest_primes`, and `seq_gen`. These functions can easily be derived from existing sequential code with minor stylistic modifications. (In fact, it is straightforward to compose a sequential prime generation application using these four functions.)

The evolving set of primes is denoted by variable `p` which initially has only the first two primes (returned by `initial_primes( 2 )`) and at each subsequent step combines the current set with the new set found using function `rest_primes( p, low, high )` where `low` is the maximum of the current set and `high` is its square.

Function `rest_primes` is a Lucid function that divides the range `lo` through `hi` into fixed intervals of size `INTERVAL`, finds primes in each of these intervals using `seq_gen`, and combines these primes into the result using `linear_tree`.

The main source of parallelism is the simultaneous invocation of `seq_gen` with `p`, the current set of primes, for the different intervals. This parallelism is implicit in the way the primes of each interval is combined using function `catenate` in `linear_tree`. Structurally, function `rest_primes` is similar to function `mergesort` that we saw earlier.

This application kernel shows how GLU can be used to express iteration and tournament parallelism within each iteration.

4.2 Matrix Multiplication

The matrix multiplication application kernel illustrates the use of `cubic_tree` dimensionally-abstract function to express three-dimensional tournament parallelism.

Given two matrices A of size p by q and B of size q by r, compute $C = AB$ where size of C is p by r such that

$$C_{i,j} = \sum_{k=1}^{q} A_{i,k} B_{k,j}.$$

The standard algorithm to compute matrix product has a complexity of $O(n^3)$ where n is the number of multiplications.

The standard algorithm can be expressed as follows. Let A and B be two n by n matrices where n is a power of two (without loss of generality). We can divide each of A and B into four $n/2$ by $n/2$ matrices and express

4.1 Prime Number Generation

The problem is to find all prime numbers between 1 and N. The solution is essentially iterative. Starting with an initial set of primes (which we will call p_0), generate the next set of primes between $\max(p_0)+2$ and $\max(p_0)^2$. The target range is divided into fixed intervals and primes are simultaneously found in each of those intervals using a sequential algorithm using the current set of primes. The primes from each of these intervals are combined to form the the new set of primes which are used to find more primes until all primes no greater than N are found.

```
struct list {
  int x;
  int v[x];
};
typedef struct list *LIST;

int display( LIST );
int max( LIST );
LIST initial_primes( int );
LIST seq_gen( LIST, int, int );
LIST catenate( LIST, LIST );

p asa ( max(p) >= N )
  where
    p = initial_primes( 2 ) fby catenate( p, rest_primes( p, low, high ) );
    low = max( p );
    high = low*low;
    rest_primes( p, lo, hi ) = linear_tree.s( catenate, segment, nsegment )
      where
        dimension s;
        segment = seq_gen( p, l, h );
        l = lo fby.s l + INTERVAL;
        h = if( l + INTERVAL > hi ) then hi else l + INTERVAL fi;
        nsegment = ( hi - lo ) / INTERVAL;
      end;
  end
```

imperative function and no state is kept at any executor such that the result of executing an imperative function depends on that state. Each executor blocks on the generator pending the generation of a ripe function. When an executor is given a ripe function to execute, it unblocks, executes the function, returns the result to the generator, and blocks again.

For each GLU program, the generator/executor abstract architecture is instantiated resulting in a specific program architecture whose generator evaluates the Lucid part of the program and executes certain imperative user-defined functions and whose executor applies certain other imperative user-defined functions to their arguments. For the mergesort program, all imperative functions except `sort` will be executed locally by the generator itself; only `sort` is granular enough to be executed by one of the executors.

3.2 Compiling Abstract Program Architectures

Once a GLU program has been mapped to an abstract architecture, the resulting abstract program architecture has to be realized on specific conventional parallel computers. Since most conventional parallel computers have a process-based operating system such as Unix augmented with some message-passing capability, the compilation to a particular target computer is fairly straightforward. Basically, the generator and executor manifest as separate executables. The generator process runs on one processor and several executor processes run on their own processors. While the executors do not communicate with each other, they all communicate and synchronize with the generator.

In the GLU system we have implemented, we use an asynchronous remote procedure call facility to support interactions between the generator and executors. The facility is at a sufficiently high level such that the nature of the underlying architecture (shared memory versus distributed memory) is hidden. It is also possible to use systems such as Linda [4] or PVM [8] or Active Messages [9] to provide this support.

4 Application Kernels in GLU

Having briefly described the GLU programming system, we now illustrate the expressiveness of GLU in two ways: in being able to succinctly express granular parallelism and in being able to reuse existing sequential code. For this, we use two kernel applications: prime number generation and block matrix multiplication.

We illustrate a purely demand-driven implementation of eduction by evaluating the mergesort program described earlier (page 10). Consider a list of m elements in dimension `seq` denoted by `L`. When `mergesort.seq( L, m )` is invoked, it causes `linear_tree.seq( merge, L, m )` to be invoked.

From the definition of `linear_tree` (page 7), it causes `b asa.t i == m` to be demanded at point `seq=0`. Demand for the value of the expression proceeds as follows: first, `i` at point `t=0,seq=0` is demanded which from the definition of `i` is 1. If the `m` at point `seq=0` is 1, then the left hand side of the `asa` expression is demanded at that point, i.e., `b` at point `t=0,seq=0`. If not, `i` at point `t=1,seq=0` is demanded which is the value of `i` at point `t=0,seq=0` times 2. Again, the resulting value of `i` is compared to that of `m` at point `seq=0`. The process continues until the comparison succeeds which will be at point `t = T` where T = $\log_2$ `m`. This will cause `b` to be demanded at point `t=T,seq=0`.

The definition of `b` is such that this demand will result in `merge( b, next.seq b)` at point `t=T-1,seq=0` which, in turn, will cause `b` to be demanded at points `t=T-1,seq=0` and `t=T-1,seq=1`. These demands will result in further demands for `b` at points `(t=T-2,seq=0)`, `(t=T-2,seq=1)`, `(t=T-2,seq=2)`, and `(t=T-2,seq=3)`.

Eventually, demands for `L` at points `seq=0`, `seq=1`, ..., `seq=m` will be issued. Since these demands can be immediately satisfied, the results will be processed by function `merge` applied to pairs of elements of `L` producing a sorted list of pairs. When these list of pairs is produced, function `merge` will again be applied pairwise to produce a list of sorted lists of size 4. This process will continue until a single sorted list of size m is produced.

Generator/Executor Abstract Architecture

The generator/executor abstract architecture consists of a single generator and multiple executors. The generator is responsible for all aspects of eductive evaluation except for execution of certain imperative user-defined functions. This is done at the executors by sending them the names of functions and their arguments and receiving the results of function applications.

Coarse grain parallelism is exploited by keeping several executors simultaneously busy. The extent to which parallelism can be exploited depends on how rapidly arguments to imperative functions can be generated by the generator and how rapidly these arguments can be transported to an executor relative to the time taken to execute the ripe functions at the executor.

Load balancing is dynamic. Each executor is capable of executing any

3 Compiling to Conventional Parallel Computers

The process of generating parallel executables for conventional parallel computers from a GLU program consists of two distinct steps.

- *Step 1.* Mapping GLU program to an abstract program architecture.
- *Step 2.* Compiling the abstract program architecture to a specific conventional parallel computer.

The executables produced by Step 2 when executed on the appropriate parallel computer would exploit coarse-grain parallelism inherent in the program. We consider each of the steps below.

3.1 Generating Abstract Program Architectures

For each GLU program, there are several possible abstract program architectures each of which constitute virtual machines for executing the GLU program. Each program architecture embodies a model of computation that unravels implicit coarse-grain parallelism in the associated program.

Currently, the GLU programming system supports a particular abstract program architecture template that we refer to as the *generator/executor abstract architecture.* We describe this abstract architecture next starting with the model of computation that it embodies.

Model of Computation

A GLU program can be given meaning mathematically without appealing to operational notions. While a GLU program does not prescribe a model of computation, its mathematical semantics has primacy over all possible models of computation that can be used to derive its meaning operationally.

The model of computation embodied by the generator/executor abstract architecture is called *eduction* [1]. Eduction corresponds to a lazy evaluation strategy for Lucid (and GLU) programs. The strategy is typically implemented using demand-driven execution although it is also possible to use a combination of demand-driven and data-driven execution [6].

Using a lazy model of computation such as eduction means that there is no superfluous computation. However, exploitation of parallelism is more conservative when compared to its eager counterpart. Also, eduction, when implemented using demand-driven execution incurs the overhead of demand propagation in evaluating programs.

```
struct list { int length; ELEMENT elts[length]; }
typedef struct list *LIST;

\\ External imperative function prototype definitions
LIST merge( LIST, LIST );

\\ Lucid composition
mergesort.s( a, l ) = b asa.t size >= l
  where
    dimension t;

    size = 1 fby 2*size;
    b = a fby.t merge( lparent.s b, rparent.s b );
  end;
```

Note that the GLU program has three parts: The first defines the external data structures, the second declares the function prototype description, and the last is the dependency specification that appeals to the external imperative function `merge`.

We can further refine this program by applying the `linear_tree` dimensionally-abstract function introduced earlier.

```
\\ External data structure definitions
typedef int ELEMENT;
struct list { int length; ELEMENT elts[length]; }
typedef struct list *LIST;

\\ External function prototype description
LIST merge( LIST, LIST );
int  listsize( LIST );

\\ Lucid composition
mergesort.s( a, l ) = linear_tree.s( merge, a, l )
```

The above is a compact expression of parallel mergesort in which the hardest-working part i.e., merge, is specified imperatively in function `merge`.

We have seen how coarse-grain dataflow programs can be developed using GLU. Next, we briefly discuss how such programs can be compiled to execute in parallel on conventional parallel computers.

2. The program makes use of the `aux` dimension to retain sorted sublists and of the `upon` operator to merge two sorted lists.

The fine-grain parallelism inherent in the above program cannot be readily and effectively exploited on conventional parallel computers. Thus, it is important to granulate parallelism expressible by Lucid. This is precisely what GLU allows. Consider the following GLU program.

```
LIST merge( LIST, LIST );

mergesort.s( a, l ) = b asa.t size >= l
  where
    dimension t;

    size = 1 fby 2*size;
    b = a fby.t merge( lparent.s b, rparent.s b );
  end;
```

The main difference between this program and the Lucid version is that `merge` is an externally-specified function that accepts two arguments each of which are externally-defined lists and returns a sorted externally-defined list. The function `merge` as far as the rest of the program is concerned is simply a pointwise function that operates on externally-defined data structures (LIST) and returns an externally-defined data structure (LIST). It is worth observing that while the Lucid program used the `aux` dimension to retain sorted sublists, the GLU program uses an external data structure to do so. Furthermore, the parallelism is coarse grain because it is at the level of merging two externally-defined lists instead of being at the level of comparing elements of two lists (as it is with the Lucid program.)

Thus, a GLU program consists of a top-level Lucid composition that implicitly expresses different kinds of parallelism and a set of externally-defined functions appealed to by the composition. By having computations expressed as imperative functions, it is possible to reuse the considerable amount of existing working software in various application domains written in languages like C and Fortran. For the purposes of this discussion, we will assume that the imperative language is C.

The above program can be further refined as follows.

```
\\ External data structure definitions
typedef int ELEMENT;
```

2.3 Granulating Parallelism

Consider a Lucid program for sorting using the mergesort algorithm. In mergesort, a list of elements (initially, singletons) are merged into a list of pairs where each pair is in sorted order. At each successive stage, list of sorted sublists are merged into list of half-as-many but twice-as-large sorted sublists until the resultant sublist is the entire list.

```
mergesort.s( a, l ) = r0t8.aux,s b asa.t size >= l
  where
    dimension t, aux;

    size = 1 fby 2*size;
    b = a fby.t merge.aux( lparent.s b, rparent.s b );
    merge.i( x, y ) = if xx <= yy then xx else yy fi
                      where
                        xx = x upon.i xx <= yy;
                        yy = x upon.i xx > yy;
                      end;
    lparent.z( c ) = c @.z (2 * #.z);
    rparent.z( c ) = c @.z (2 * #.z + 1);
  end;
```

The dimensionally-abstract function `mergesort` takes a list `a` of `l` elements in the `s` dimension and returns a sorted list in the same dimension. The definition of `b` initially is the unsorted list; at each successive stage (in the `t` dimension), it is the result of merging pairs of elements of the list (which are really sublists). The merging is done using function `merge` and the crucial point is that it accepts two sublists as its arguments and returns a single sorted list that varies in the `aux` dimension which is orthogonal to the `s` dimension. Thus at each stage, there are half as many elements (or sublists) in the `s` dimension and twice as elements per sublist which is defined over the `aux` dimension. Finally (when there is only one sublist with the same size as the original unsorted list), the sorted list `b` which is defined over the `aux` dimension is converted into a list that is defined over the `s` dimension using the `r0t8` operator.

This example illustrates the use of Lucid to express computation and therein parallelism. There are two points to observe:

1. The parallelism in the simultaneous merging of pairs of sublists is essentially fine grain (at the level of comparison of two elements).

The variable `c` is defined to be the value of `b` at the point in the `t` dimension where the value of `i` equals n (which we assume to be a power of 2). Variable `i` is defined as 1 at `t`-dimension point 0, 2 at point 1, 4 at point 2, and so on. The point in the `t`-dimension when `i` will equal n is actually $\log_2(n)$.

Variable `b` is initially the same as `a` for all points in the `x` dimension. At each successive point in the `t` dimension, the value of `b` at a point k in the `x`-dimension if the value returned by function `f` applied to the value of `b` at points $2k$ and $2k+1$ in the `x`-dimension and at the previous point in the `t`-dimension.

The following is an equivalent mathematical definition of `b`.

$$\mathtt{b}_{x=k,t=l} = \begin{cases} \mathtt{a}_{x=k} & \text{if } \mathtt{t} \text{ is } 0 \\ \mathtt{f}(\mathtt{b}_{x=2k,t=l-1}, \mathtt{b}_{x=2k+1,t=l-1}) & \text{otherwise} \end{cases}$$

Tree computations occur frequently enough that we can define a dimensionally-abstract function `linear_tree` to express one-dimensional tree computations.

```
linear_tree.z( f, a, n ) = b asa.t i == n
  where
    dimension t;
    i = 1 fby.t 2*i;
    b = a fby.t ( f( b, next.z b) @.z (2 * #.z) );
  end;
```

Expression `linear_tree.x( f, a, n )` expresses precisely the same computation as given in the earlier example. And `linear_tree.x( add, a, n)` computes the sum of the first n values of `a` in the `x` dimension using a binary summation tree. At each level of binary tree, pairwise additions can occur simultaneously with the degree of data parallelism being $n/2$ initially and decreasing geometrically by 2 with each additional level.

Similarly, it is possible to define dimensionally-abstract functions `planar_tree` and `cubic_tree` to express tree computations in two and three dimensions respectively. As an example of their use, expression `planar_tree.x,y( max, a, n*n)` computes the maximum of n^2 values (at points given in the plane defined by the first n points in dimension `x` and `y`) using a two-dimensional binary tree or pyramid. Each level of the pyramid, comparisons in four value groups can occur in parallel with the degree of data parallelism being $n * n/4$ initially and decreasing geometrically by four with each additional level.

The computation of successive values of `a` in the `x` dimension has implicit parallelism due to pipelining. While `c` at some point i in dimension `x` is being computed, `b` at point $i-1$ in dimension `x` is being computed and `a` at point $i-2$ in dimension `x` is also being computed.

2.2.2 Data Parallelism

Simply speaking, data parallelism is the simultaneous processing of different data using the same function. Data parallelism is the principal source of massive parallelism in most scientific computations. Lucid, being a multidimensional language, is well suited to expressing such parallelism as the following example illustrates.

```
a = b fby if boundary
          then B
          else ( prev.x a + next.x a + prev.y a + next.y a ) / 4
          fi;
```

In this example, variable `a` can be thought of as a temporal stream of planes. Initially, the value of each element of `a` (as given by the implicit contexts `x` and `y`) is the value of the corresponding element of `b`. At each successive time step, the each value of `a` is either the boundary value `B` if the point of the value is at a boundary or it is the average of the four neighboring values of `a` from the previous time. Note that the four neighbors can be accessed by manipulating the `x` and `y` contexts using operations `prev` and `next`.

2.2.3 Tournament Parallelism

Tournament parallelism arises in tree computations as logarithmically decreasing data parallelism at each level of the tree [2]. (It is closely related to parallelism found in "divide-and-conquer" algorithms.) In Lucid, tournament parallelism is an example of dimensional parallelism as illustrated by the following example.

```
c = b asa.t i == n
    where
      dimension t;
      i = 1 fby.t 2*i;
      b = a fby.t ( f( b, next.x b) @.x (2 * #.x) );
    end;
```

The value of `a` at a given point in the `x` dimension is the value of function `nxt` given arguments `b` and `n` at the same point. Function `nxt` is the value of `b`, n points from the current point in the `x` dimension.

In Lucid, user-defined functions like operations can be dimensionally abstract. For example, it is possible to define the function `nxt` such that it works in any dimension rather than just in the `x` dimension.

```
a = nxt.x( b, n )
where
  nxt.z( c, n ) =  c @.z (#.z + n );
end
```

Next, we shall see how dimensionally-abstract functions can be used as "templates" for implicitly expressing parallelism.

2.2 Implicit Parallelism in Lucid Compositions

Almost all parallelism expressed in Lucid can be characterized as dimensional parallelism because they arise out of multidimensional Lucid compositions. (Since Lucid is also a first-order functional language, standard functional parallelism can be expressed orthogonal to dimensional parallelism.)

We illustrate various kinds of dimensional parallelism by way of simple examples.

2.2.1 Pipelined Parallelism

Pipelined parallelism arises when a stream of data is being processed by a "pipeline" of functions where each function of the pipeline is simultaneously processing a different part of the data stream. Also known as stream parallelism, pipelined parallelism is naturally expressible in dataflow languages. In Lucid, it is a simple instance of dimensional parallelism as the following example illustrates.

```
a where
  a = 0 fby.x f( b );
  b = 0 fby.x g( c );
  c = 0 fby.x h( d );
  d = 1 fby.x d + 1;
end
```

or `false` at that point.

If the operation is a dimensional operation, then the value of the LHS variable at a given point is the result of the operation applied to values of the operand terms at points defined by the operation.

The two basic dimensional operations are `@` and `#`. The expression `a @.x n` refers to the value of `a` at the $\mathtt{n}^{th}$ point in dimension `x`. The expression `#.x` refers to the "current" implicit point of evaluation in the `x` dimension.

Other dimensional operations can be defined in terms of `@` and `#`. For example, the dimensional operation `fby.x` (which should be read as "followed by in the `x` dimension"), as in `a fby.x b`, can be defined as follows:

```
if( #.x == 0 ) then a @.x 0 else b @.x (#.x - 1) fi;
```

In the equation, `a = b fby.x c`, for all points where the `x` dimension position is 0, the value of `a` is the same as the value of `b` at that point. For all other points, the value of `a` is the value of `c` at the preceding point (in the `x` dimension).

Also, dimensional operation `asa.x` (which should be read as "as soon as in the `x` dimension"), in expression `e asa.x p`, can be defined as follows:

```
e asa.x p = a
where
  a = if( p @.x #.x ) then e @.x #.x else next.x a fi
      where
        next.x a = a @.x (#.x + 1);
      end;
end
```

In the equation, `a = e asa.x p`, the value of `a` at all points in the `x` dimension is the value of `e` at a point in the `x` dimension such that the value of `p` at that point is `true` and the value of `p` at each of the preceding points in the `x` dimension is `false`.

If the RHS is a user-defined function, the value of the LHS variable is the same as the value of invoking the RHS function on its argument expressions using call-by-need semantics.

```
a = nxt( b, n )
where
  nxt( c, n ) = c @.x (#.x + n );
end
```

consists of two distinct parts: a Lucid part that specifies the program composition and a C part that defines various data types and data functions referred to in the Lucid part.

2.1 Lucid circa 1994

The language Lucid has evolved from a temporal dataflow language [10] to a multidimensional dataflow language [3]. It is the only dataflow language in which multidimensional data structures are implicit which has significant consequences on its ability to express different kinds of parallelism as we shall see later in this chapter. We briefly review the essential aspects of Lucid circa 1994.

A Lucid program is a structured set of equations where each equation defines a variable. Associated with each set of equations is one or more user-defined dimensions. These orthogonal dimensions, which are infinite in extent, define a multidimensional space in which each variable denotes a scalar value at each point in the space. Thus, a Lucid program computes values of specific variables at specific points in the nested multidimensional space that it defines. Computation of each desired value requires the computation of other values at various points as specified by the equations of the Lucid program.

Let us briefly consider the syntax of Lucid. The left-hand-side (LHS) of an equation is a variable. The right-hand-side (RHS) of each equation is a term which can be a constant, a variable, an operation applied to other terms, or a user-defined function whose arguments are other terms.

If the RHS is a constant, the value of the LHS variable at all points in the enclosing multidimensional space is that constant.

If the RHS is a variable, the value of the LHS variable is the same as that of variable on the RHS at all points in the enclosing multidimensional space.

If the RHS is an operation applied to other terms, the value at each point of the LHS variable is given by the application of the operation to values of the operand terms at appropriate points in the enclosing multidimensional space.

If the operation is a data operation such as `+`, or `<`, the value of the LHS variable at a given point is the result of the operation applied to the appropriate values of the operand terms at the same point. For example, the equation `a = if p then b else c fi` defines `a` at each point to be either the value of `b` or `c` at that point depending on whether the value of `p` is `true`

If such a language could be efficiently implemented, it would considerably alleviate the difficulties of programming such computers.

GLU (short for Granular Lucid) is a high-level dataflow language for programming conventional parallel computers. It is a coarse-grain version of the multidimensional dataflow programming language Lucid by extending Lucid in two simple ways: user-defined functions are specified in a foreign language (such as C) and values are of foreign types (such as C data types) [5, 3].

Programming a conventional parallel computer with GLU consists of three stages:

1. Develop a coarse-grain dataflow program using GLU in which parallelism is implicitly expressed.
2. Select an appropriate abstract architecture to which the program is automatically mapped.
3. Compile the abstract program architecture to a specific conventional parallel computer.

GLU programs are highly portable because architectural dependencies are hidden from the program itself and captured in Stages 2 and 3. GLU programs can largely be derived from existing sequential applications since user-defined functions can reuse procedural code with only nominal modifications.

In addition to describing the GLU system in this paper, we also discuss how well GLU addresses two important issues relating to coarse-grain dataflow programming of conventional parallel computers, namely, expressiveness and efficiency. Expressiveness of GLU is the extent to which diverse parallel programs can be easily expressed and the extent to which existing sequential code can be reused with only nominal modifications. Efficiency of GLU is measured by the extent to which inherent coarse-grain parallelism in GLU programs is exploited on conventional parallel computers. We address these issues by using well-understood application kernels (sorting, prime number generation, and matrix multiplication) and a small-scale yet ubiquitous parallel computer, namely, a workstation cluster.

2 GLU Model of Programming

GLU is a hybrid programming model that combines the multidimensional dataflow language Lucid and the procedural language C. A GLU program

Coarse-Grain Dataflow Programming of Conventional Parallel Computers*

R. Jagannathan
Computer Science Laboratory
SRI International
Menlo Park, California 94025
U.S.A.

Abstract

Granular Lucid (or GLU) is a coarse-grain dataflow language for programming conventional parallel computers. It is based on Lucid (circa 1994) which is an implicitly parallel, multidimensional dataflow language. A GLU program is a Lucid program with imperatively-defined data functions and data types.

In this paper, we briefly describe a system for coarse-grain parallel programming based on GLU. We discuss the expressiveness of GLU in composing different kinds of parallel programs. We also discuss the efficiency with which parallelism in GLU programs can be exploited on conventional parallel computers.

1 Introduction

One of the main advantages of dataflow computers is that they are easy to program since they directly support high-level dataflow languages. Conventional parallel computers, on the other hand, embody low-level programming models based on communicating sequential processes, thus forcing the programmer to deal with operational issues that have nothing to do with the application itself. What is desirable is a high-level language based on dataflow programming principles for programming conventional parallel computers.

*Draft of a chapter to appear in IEEE Computer Society's upcoming monograph titled "Advanced Topics in Dataflow Computing and Multithreading" edited by L. Bic, G.Gao, and J-L. Gaudiot.