Toward a Scalably Efficient Parallel Implementation of a Hybrid Intensional Language*

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

E-mail: jagan@csl.sri.com
WWW: http://www.csl.sri.com/jagan

Abstract

GLU is a hybrid intensional language for programming conventional parallel computers in which parallel programs are constructed by composing sequential, granular C functions using Lucid, a purely intensional language. Parallelism implicit in the Lucid specification is exploited at runtime using *eduction*, an intensional model of computation.

We describe a basic eduction-based parallel implementation of GLU consisting of a single master process and multiple stateless worker processes. Using the CM-5, we show that this GLU implementation has limited performance scalability, especially when the granularity of parallelism is not sufficiently coarse. We show how the scalability can be improved by decentralizing some state from the master to the workers. We then propose a new eduction-based parallel implementation of GLU using multiple masters that is designed to be more scalably efficient than either of the single-master parallel implementations.

1 Introduction

GLU (acronym for Granular Lucid) is a hybrid intensional language for programming conventional parallel computers[1, 2]. A GLU program consists of two parts — computation and composition. The computation part consists of a set of imperatively-specified sequential (e.g., C) functions and the composition part consists of an intensional program (using the language Lucid[3]) that uses the C functions as building blocks to compose a parallel program.

Parallelism is implicit in the specification given in Lucid which is sufficiently rich to express different kinds of parallelism including data, function, and stream parallelism[1]. The granularity of parallelism depends on the complexity of the sequential functions used in the GLU program. Since most conventional parallel computers are suited for exploiting a relatively coarse granularity of parallelism, we will assume that granularity of parallelism in GLU programs is correspondingly coarse.

Latent parallelism in GLU programs can be discovered at runtime by an intensional model of computation called *eduction*[4] from the Lucid part of the program. Eduction corresponds to tagged demand-driven evaluation — one that avoids wasteful computation at the expense of evaluation speed. However, when evaluating GLU programs with coarse granularity of parallelism, it is far more important to avoid wasteful computation than to improve evaluation speed of the Lucid part.

There are several ways in which the eduction model can be implemented on a typical conventional parallel computer. Each of these ways can be thought of as parallel programs architectures to which GLU programs are mapped. The resulting parallel programs can then be compiled to target-specific executables. This is shown in Figure 1.

The hybrid philosophy of GLU where computation is separated from composition or coordination is similar to other parallel programming languages such as PCN[5] and Linda[6]. It is unique in its use of intensionality[7, 8] to express the compositional aspects of parallel programs[3]. Being able to specify

*To appear in *Proceedings of the Second International Conference on High Performance Computing, Tata McGraw Hill, New Delhi, December 1995.*

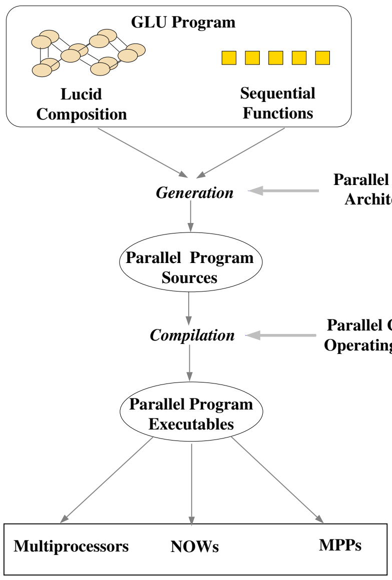

Figure 1: GLU System Architecture

parallelism intensionally at a high level of abstraction unburdens the programmer from many aspects of explicit parallelism discovery and management and helps focus the attention on the algorithmic aspects of the parallel program. GLU has been successfully used in rapidly developing implicitly-parallel versions of sequential applications in application domains such as structural stress analysis, video compression, 3D rendering, image processing, protocol processing, and large-software maintenance tools[9, 10, 11, 12].

We first briefly describe the GLU language with a prototypical data-parallel example. We then describe the single-master multiple stateless-worker implementation of GLU and consider its performance scalability on a CM-5 for different granularities of parallelism. We show how the performance scalability can be improved by selectively shifting program-specific state from the master to the workers. Finally, we propose a new eduction-based implementation for GLU based on multiple masters designed to exhibit better scalability in performance when compared to either single-master implementations.

2 A GLU Example

A GLU program consists of a composition part as a Lucid program and a computation part as set of C functions that the composition part uses. Lucid is an intensional declarative language — the intensionality allows expression of massive amounts of parallelism in a succinct manner, and its declarative nature makes this implicit.

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 can be defined at each point in space, possibly in terms of values of other points. Parallelism is present when it is possible to simultaneously compute values of a variable at *different points* in the multidimensional space and when it is possible to simultaneously compute values of *different variables* at the same point.

Consider the GLU program in Figure 2 to perform a 2-D successive relaxation (SR). The prototype definitions of the C functions `combine`, `relax`, `getband`, and `gridin` are given. The functions themselves are defined elsewhere. The C datatypes `GRID` and `BAND` are also defined in the included file, `sr.h`. Function `combine` accepts four data structures of type `GRID` and returns a single data structure of the same type. Function `relax` accepts a data structure of type `GRID` and four data structures of type `BAND` and returns a data structure of type `GRID`. Function `getband` returns a data structure of type `BAND` when provided with a data structure of type `GRID` and a value of type `DIRECTION`. Function `gridin` returns three user-provided inputs — data structure of type `GRID` and two integers. These inputs are used to define variable `igrid`, `nsubgrid`, and `niter` where `igrid` denotes the initial grid of values, `nsubgrid` denotes the number of subgrids to be used, and `niter` denotes the number of iterations of successive relaxation.

The program operates in the `time` dimension. It returns the value of variable `grid` at `time`-point `niter-1`. The value of `grid` at a specific `time`-point can be generated from the subgrids (denoted by `subgrid`) that span the `x` and `y` dimensions using a pyramid composition (as defined using the `planar_tree` builtin function). At the base of the pyramid are `nsubgrid` subgrids denoted by the variable `subgrid` that spans the `time`, `x` and `y` dimensions. At a given `time`, `x`, `y` point, variable `subgrid` denotes the result of applying imperative function `relax` to the same variable at the same `x` and `y` point but the previous `time` point along with borders obtained from each of the neighboring grids at the previous `time` point. Each interior plane of the pyramid is generated using multiple invocations of the imperative function `combine` on sets of four adjacent subgrids. Each such plane consists of one quarter as many subgrids as the level below with each subgrid

```
/* C datatype definitions */
#include "sr.h"
/* C function prototype definitions */
GRID combine( GRID, GRID, GRID, GRID );
GRID relax( GRID, BAND, BAND, BAND, BAND );
BAND getband( GRID, DIRECTION );
(GRID, int, int) gridin();
/* Lucid Composition */
grid asa.time #.time + 1 == niter where
  grid = planar_tree.x,y( combine, subgrid, nsubgrid ) where
    dimension x,y;
    subgrid = isubgrid fby.time relax( subgrid, aboveband, belowband, leftband, rightband );
    isubgrid = sel( igrid, #.x, #.y );
    aboveband = if( #.y == 0 ) then outband else prev.y getband( subgrid, SOUTH ) fi;
    belowband = if( #.y == nsubgrid - 1 ) then outband else next.y getband( subgrid, NORTH ) fi;
    leftband = if( #.x == 0 ) then outband else prev.x getband( subgrid, EAST ) fi;
    rightband = if( #.x == nsubgrid - 1 ) then outband else next.x getband( subgrid, WEST ) fi;
    outband = getband( subgrid, BOUNDARY );
  end;
  ( igrid, nsubgrid, niter ) = gridin();
end
```

Figure 2: GLU program for two-dimensional SR

being quadruple the size of the subgrid below. The root of the pyramid is the desired value of `grid`.

Let us consider how the above program is eductively evaluated. First, the program is demanded at `time`-point 0. The value of the program is the value of `grid` when the `time`-point added to 1 equals the value of `niter`. Therefore, `grid` is demanded at `time`-point `niter-1`. The definition of `grid` states that it can be obtained by demanding a 2-D pyramid composition (using `planar_tree`) of `subgrid` spanning the `x` and `y` dimensions at the same `time`-point (`niter-1`). The definition of `subgrid` states that it is given by `isubgrid` at `time`-point 0 and by the application of the imperative function `relax` at each successive `time`-point with arguments `subgrid` at the previous `time`-point (and same `x` and `y` points), and the borders of the four neighboring subgrids at the previous `time`-point. Note that the neighboring subgrids can be named simply by manipulating the `x` and `y` points as appropriate. In this program, parallelism is exploited in the simultaneous computation of the variable `subgrid` at the various `x` and `y` points even though each subgrid at a given `x` and `y` point is computed sequentially over the `time` dimension. This parallelism is *implicit* in the definition of `SR`, and its degree is proportional to the value of `nsubgrid`.

3 Single-Master/Multiple-Worker Implementations

Figure 3 shows the architecture of a generic single-master, multiple-worker implementation.

The master is responsible for *sequentially* interpreting the Lucid part of the GLU program using eduction. The stateless workers are responsible for executing imperatively specified procedural functions as and when indicated by the master.

The master consists of three logical components — the suspended function manager (SFM), the local function executor (LFE), and the variable value manager (VVM). The SFM is responsible for demand propagation; the VVM is responsible for retaining values of variables that have been already computed; and the LFE is responsible for applying local functions and operations to their arguments to produce a result.

Program execution begins by a demand for a result value (`?t`) being made to the SFM. If the result is a term (i.e., a simple expression), the demand is processed at the SFM itself as described later. If the result is a variable, the VVM is checked to see if the demanded value (`?v`) has already been computed, in which case it (`*v`) is returned thus satisfying the demand. If the value has not been computed, the term that defines the variable is demanded (`?t`) at the SFM.

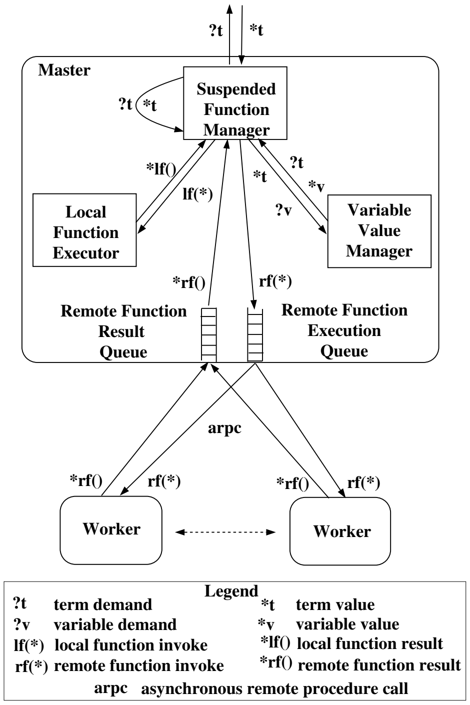

Figure 3: Single-Master/Multiple-Worker Implementation Architecture

When a term is demanded at the SFM, it causes the creation of a suspended operation (or function), which is essentially the operation (or function) together with a set of placeholders for each argument of the operation. Each argument (which corresponds to a term or a variable) is demanded (`?t`) at the appropriate point as determined by the operation itself. For pointwise operators, the point of the operand values is the same as the point of the result being demanded. For non-pointwise (Lucid) operators, the point of the operand values is modified suitably to reflect the semantics of the operator. Thus, processing a demand at the SFM results in a suspended operation to be created and further demands to be propagated to the SFM when a term value is being demanded and to the VVM when a variable value is being demanded. (A demand is immediately satisfied if the argument being demanded is a constant.)

When a demand for a value is processed at the VVM, a determination is made of whether the value has already been computed. If so, the value is returned to the demander, which is a suspended operation in the SFM. If not, the name of the value is recorded in the VVM with its status set to "being computed" and the demand for its defining term is sent to the SFM, which is processed as described above. Further demands for the same value prior to its computation would cause them to be queued until the value is computed.

Values are returned to the SFM either by finding them in the VVM (`*v`) or when the arguments being demanded are constants (or inputs). When a value is demanded, the return address (the address of the specific suspended operation) is retained with the demand so that when the value is returned, it is inserted in the appropriate place.

When the required values are available for a suspended operation or a suspended local function, it (`lf(*)`) is unsuspended and sent for execution to the LFE. When the result is produced by the LFE, it (`*lf()`) is sent to the return address pointed to by the suspended operation, which could be another suspended operation in the SFM or it could be the value of a variable in the VVM. In the latter case, all pending demands for that variable value at the VVM are satisfied and returned to their respective return addresses.

When the required values are available for a suspended remotely executable function, it (`rf(*)`) is unsuspended and queued in the remote function execution queue (RFEQ). When the result of remote function execution (`*rf()`) is placed by a worker in the remote function result queue (RFRQ), it is returned to the SFM to satisfy a demand issued by a suspended operation or function.

The workers block on the RFEQ for an executable remote function consisting of the function name and arguments. Each worker is unblocked when it is assigned an executable remote function (`rf(*)`) that it executes in its own address space. The result of the remote function execution (`*rf()`) is placed by the worker in the master's RFRQ.

The communication and synchronization mechanism used by the workers to interact with the master is using asynchronous remote procedure calls.

The main source of parallelism is in being able to keep the multiple workers simultaneously busy with remote function executions. This depends on how much parallelism is inherent in the program being evaluated. It also depends on the communication latencies to transport remote executable functions (i.e., function names and arguments) and results of remote function executions relative to the time taken to execute remote functions. The ratio of computing time for remote function execution to communication times to transport arguments and result is referred to as granularity of parallelism.

The master performs three activities that are sources of sequentiality:

1. Eductive evaluation of the Lucid part of the GLU program as captured in the operations of the SFM and VVM
2. Execution of locally executable imperative functions in the LFE
3. Communicating with the workers through RFEQ and RFRQ

The time spent by the master on the above three activities should be kept to a minimum. The program being evaluated plays a significant role in determining the size of the overhead. If the Lucid part is complex, it might entail considerable time spent in eductive evaluation. If the program has several local functions, time could be spent in the master in local function executions. And if the arguments and results being exchanged between the master and workers are large, the time spent by the master in the communication and synchronization aspects via RFEQ and RFRQ management could be sizable.

In this implementation architecture, any worker can execute any executable remote imperative function. This is possible because workers are stateless — they do not maintain any program-dependent state across function executions. Thus, when a master sends an executable imperative function to a worker, the name of the function along with its arguments must be passed. The worker, upon completion, returns the result of the function execution to the master. And it is ready to execute another function provided by the master.

In the successive relaxation program example, we assume that all imperative functions except `relax` are locally executed in the master. (Whether an imperative function executes locally or remotely is controlled by the programmer at run-time.) When remotely executing each invocation of `relax`, the master provides the worker with the arguments to `relax` — the subgrid itself and its borders. The worker returns the new subgrid as the result.

3.1 Stateful Workers

In this variation of the single-master/multiple-worker implementation architecture, some of the state centrally stored at the master is distributed among the workers. In particular, instead of storing data needed by a remotely-executable imperative function in the master, it is stored instead at the worker. This is done explicitly by the programmer by introducing state into the imperative function code, and introducing new imperative functions to store and retrieve the data at the worker, from the master. The programmer also explicitly controls mapping of work (i.e., executable functions) to appropriate workers by annotating the Lucid specification.

The successive relaxation program can be changed as shown in Figure 4 to map specific invocations of functions `relax` and `getband` to specific workers. The variable `subgrid` no longer refers to a subgrid; rather it is a handle to a remotely stored subgrid that is returned by function `STORE` and used in retrieving the subgrid using function `FETCH`. All uses of `FETCH` and `STORE` are augmented with the mapping expression [#.x + #.y], which means that fetching and storage will occur in logical worker `#.x + #.y`. (Note that the # operation simply returns the position information along its qualifying dimension.) Both `relax` and `getband` are also augmented with the mapping expression `#.x + #.y`, which means they will execute on the logical worker given by that expression.

Figure 5 shows the two implementations (basic implementation or stateless worker implementation and the stateful worker implementation) of the GLU program. The main differences between the two implementations are as follows:

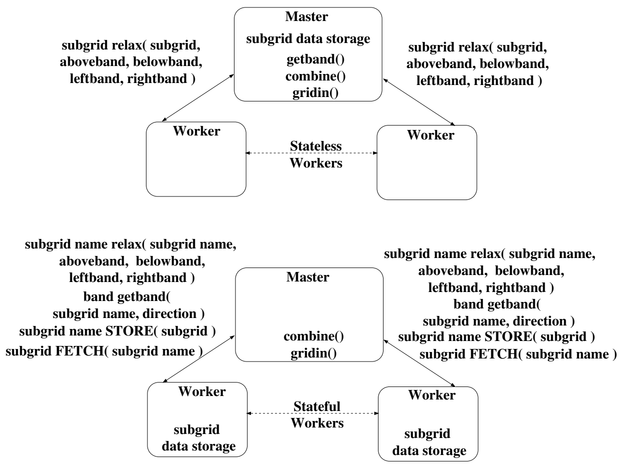

Figure 5: Two Implementations of the SR GLU Program

- The grid data is stored in the master with the stateless worker implementation whereas it is distributed across the workers with the stateful worker implementation.
- Invocations of function `relax` are executed in an arbitrary worker in the stateless worker imple-

```
grid asa.time #.time + 1 == niter where
  grid = planar_tree.x,y( combine, FETCH[#.x + #.y]subgrid, nsubgrid )
  where dimension x,y;
    subgrid = STORE[#.x + #.y]( isubgrid ) fby.time
              relax[#.x + #.y]( subgrid, aboveband, belowband,
                                           leftband, rightband );
    isubgrid = sel( igrid, #.x, #.y );
    aboveband = if( #.y == 0 ) then outband
                else prev.y getband[#.x + #.y]( subgrid, SOUTH ) fi;
    belowband = if( #.y == nsubgrid - 1 ) then outband
                else next.y getband[#.x + #.y]( subgrid, NORTH ) fi;
    leftband = if( #.x == 0 ) then outband
               else prev.x getband[#.x + #.y]( subgrid, EAST ) fi;
    rightband = if( #.x == nsubgrid - 1 ) then outband
                else next.x getband[#.x + #.y]( subgrid, WEST ) fi;
    outband = getband[#.x + #.y]( subgrid, BOUNDARY );
  end;
  ( igrid, nsubgrid, niter ) = gridin();
end
```

Figure 4: GLU program for SR with function mapping

mentation and on predetermined workers in the stateful workers variation.

- Invocations of function `getband` are executed in the master itself in the stateless worker implementation and on predetermined workers in the stateful workers variation.

- Each remote invocation of function `relax` requires transmission of relevant subgrid between the master and an arbitrary worker with the stateless worker implementation whereas this is avoided in the stateful worker implementation by storing the subgrid data in the worker itself using `STORE` at the beginning, and retrieving it using `FETCH` at the end.

4 Performance Study

To understand the relative merits of the single-master/stateless-workers implementation and the stateful workers variation, we consider the performance of the successive relaxation application kernel with three different granularities of parallelism, all on the CM-5. The problem size is 512×512 points configured as 32 subgrids of size 16×512. Three different filters are used, of sizes 11×11, 7×7, and 3×3.

Figure 6 shows the speedups obtained using the two implementation architectures.

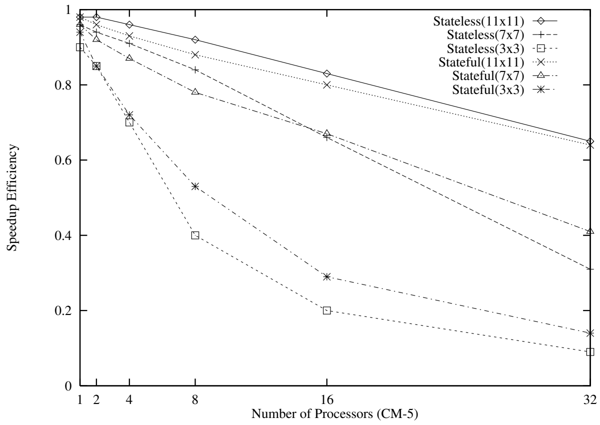

Figure 6: Performance Comparison of Stateless and Stateful Worker Single-Master Implementations

From these results, we can observe that the stateless worker implementation exhibits better performance than the stateful worker implementation with a small number of workers and that the stateless worker implementation shows performance degradation sooner than the stateful worker implementation with an increasing number of workers. With both implementations, the extent to which performance can be sustained with an increasing number of workers depends on the coarseness of the granularity of parallelism. It is worth noting that when the stateless worker implementation is superior to the stateful worker implementation, it is due to the positive effect of dynamic load distribution coupled with coarse granularity of parallelism despite the greater communication overhead.

Table 1 shows the performance of the successive relaxation kernel using stateless workers. (The results are shown in that order for each number of workers.) Although the kernel consists of several functions, only one, **`relax`**, is executed remotely. This function accepts a subgrid and borders (whose size depends on the filter size) from the neighboring subgrids and returns the new values for the subgrid by applying the filter. The average time taken to execute a remote function by the worker for filter size 11×11 is 4.0 seconds, while the average time taken to transport arguments and results between master and worker is 0.095 second. The average time taken to execute a remote function by the worker for filter size 7×7 is 1.8 seconds while the average time taken to transport arguments and results between master and worker is 0.085 second. The average time taken to execute a remote function by the worker for filter size 3×3 is 0.4 seconds while the average time taken to transport arguments and results between master and worker is 0.080 second.

From these results, we can infer that the performance of the stateless worker variation is less scalable with decreasing granularity of parallelism. While the performance scales quite reasonably for 32 workers when the granularity is coarse (in the 11×11 case), it does not scale beyond 16 workers in the 7×7 case and beyond 8 workers in the 3×3 case. This lack of scalability is caused by the extent to which the master is occupied with overhead activities, principally communication and local function evaluation, and unable to generate sufficient work for the workers. These cases are marked by high master and low average worker utilizations.

Table 2 shows the performance of the successive relaxation application kernel using stateful workers in which the subgrids are retained in the workers. The main difference to observe is in the communication overhead, which has been reduced by approximately 20% in the 11×11 case, by approximately 35% in the 7×7 case, and by approximately 50% in the 3×3 case. There is no marked change in the other sources of master overhead — the eduction overhead and the local function execution overhead.

5 A Multiple Master Implementation

The stateful worker variation of the single-master implementation reduces communication overhead by storing large data structures remotely at the workers (where they are needed) and consequently improves performance scalability. However, the implementation still requires local function evaluation and eductive evaluation to occur in the master. This has several consequences on performance scalability:

- Evaluating local functions in the master only can create unnecessary sequentiality. For example, all invocations of the relatively small grain `combine` function in the SR example is always executed in the master in sequential order even though these invocations can, in principle, be executed in parallel.
- Conducting eductive evaluation at the master means that all interaction between workers has to go through the master, thus creating a bottleneck. For example, in the SR example, all exchanges of band data between the workers flows through the master.

We propose a fully decentralized multiple master implementation for GLU programs (see Figure 7) that addresses the problems of centralized local function and eductive evaluations. Instead of one master and multiple workers, the implementation consists of multiple masters. The basic idea is quite simple — distribute eductive evaluation across the masters and perform *all* ensuing function execution within each master. We assume that each master is multithreaded in the sense that function execution can proceed within a master at the same time as eductive evaluation, and the overhead of conducting function execution is nominal.

Eductive evaluation is distributed across masters as follows. The VVM of a master i (denoted by $\texttt{VVM(}G_i\texttt{)}$) upon receiving a demand for a variable value $\mathtt{v}_c$ determines whether this value should be computed locally or at a remote master. In the first case, it proceeds

Workers	Execution Time (s)	Master Times (s) Communication	Eduction	Local Function	Master Utilization	Worker Utilization
sequential	517.4	-	-	-	-	-
	222.8	-	-	-	-	-
	59.0	-	-	-	-	-
1	523.0	11.7	1.6	7.5	4.0	97.2
	230.7	10.9	0.9	7.4	8.3	93.8
	65.1	9.8	1.2	6.4	26.5	79.1
2	263.4	11.7	1.6	7.5	7.9	96.6
	117.4	10.8	0.9	7.4	16.3	92.8
	34.6	9.9	1.3	6.1	50.0	75.9
4	133.9	11.8	1.4	7.6	15.5	95.3
	60.9	10.9	0.9	7.4	31.5	90.5
	20.9	10.1	1.4	6.6	86.3	66.1
8	70.1	12.3	1.0	8.0	30.4	92.2
	32.9	10.9	0.9	7.4	58.3	85.6
	18.0	9.6	1.4	6.5	97.0	37.9
16	38.6	12.4	1.3	8.1	56.4	85.5
	21.0	10.9	1.2	7.8	94.7	69.7
	18.0	9.6	1.2	6.5	96.2	18.9
32	24.5	12.2	1.0	8.1	87.2	69.1
	22.4	10.9	1.0	7.5	86.7	32.4
	19.0	10.3	1.2	6.4	96.4	9.2

Table 1: Detailed Performance of Successive Relaxation Kernel with Stateless Workers

.

Workers	Execution Time (s)	Master Times (s) Communication	Eduction	Local Function	Master Utilization	Worker Utilization
sequential	517.4	-	-	-	-	-
	222.8	-	-	-	-	-
	59.0	-	-	-	-	-
1	523.8	8.8	1.6	7.4	3.4	97.3
	231.4	6.5	1.2	7.4	6.5	94.6
	62.7	3.8	1.8	6.2	18.9	85.0
2	267.8	9.1	2.0	7.3	6.9	95.2
	120.4	6.5	1.2	7.1	12.2	91.9
	34.5	3.9	1.3	6.2	33.0	77.9
4	137.9	9.4	1.7	7.4	13.4	92.8
	63.6	6.6	1.1	7.1	23.3	86.7
	20.3	4.0	1.5	6.2	57.1	66.4
8	72.7	9.2	1.4	7.3	24.7	88.4
	35.5	6.6	1.2	7.2	41.9	78.5
	13.8	4.0	1.2	6.4	84.1	51.1
16	40.0	9.2	1.3	7.6	45.3	81.2
	20.7	6.6	1.2	7.1	71.9	69.0
	12.3	4.1	1.2	6.5	95.5	28.6
32	25.1	9.3	1.4	7.8	73.5	66.8
	16.7	7.0	1.0	7.2	91.4	43.8
	12.4	4.3	1.5	6.4	97.9	14.2

Table 2: Detailed Performance of Successive Relaxation Kernel with Stateful Workers

.

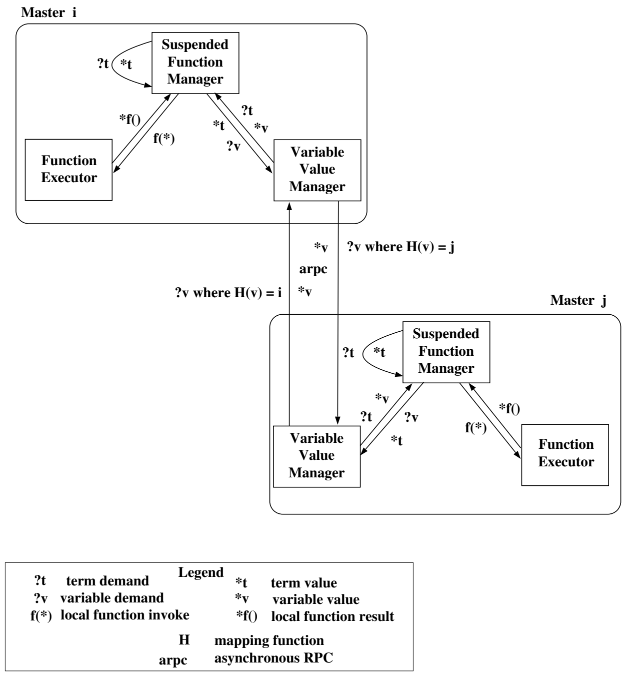

Figure 7: Multiple-Master Implementation

in the same way as the single master. In the second case, it sends the demand for the variable value to the remote master (say VVM(G_j)) where it is processed locally. When the value is available, it is returned to VVM(G_i), which returns the value to the local SFM.

There are two ways in which a VVM can determine if a variable-value-demand can be processed locally or at a remote master. One way is to hash on the variable name and point using a mapping function to compute the name of a master to which the variable value is sent. Another way is to let the mapping function be specified as part of the variable definition in the program's Lucid composition part. The first way is program-independent but it can result in uneven load distribution if the mapping function is nonuniform with respect to the program. The second way requires programmer guidance[1], which can result in very evenly balanced load distribution.

Consider the following equation from the successive relaxation program given in Figure 2.

```
subgrid[ #.x + #.y ] = isubgrid fby.time
    relax( subgrid, aboveband, belowband,
                    leftband, rightband );
```

[1] In principle, it should be possible for a decision procedure to automatically determine the mapping function of each variable through dimensionality and dependency analysis of the Lucid part of the GLU program.

The definition of `subgrid` is augmented with a mapping function, `#.x + #.y`, that determines which master should be responsible for the computation of the different subgrids. When the value of `subgrid` is demanded at the VVM of a master, it uses this mapping function to determine where to forward the demand. In this case, `subgrid` at `x`-point i and `y`-point j is computed in master $(i+j) mod\ n$ where n is the total number of masters.

When this program is executed, the subgrids and their associated evaluation are distributed across the multiple masters. The communication between masters initially consists of the dispersal of the initial subgrid values from the "home" master to the other masters and at the end consists of the collection of the final subgrid values at the home master from the various masters. Between the initial and final stages, the communication consists of exchange of subgrid borders.

The multiple master implementation, by distributing eduction, not only forces all three sources of overhead to be distributed but causes communication overhead to be reduced by avoiding unnecessary data exchanges. The extent to which the implementation will enhance performance scalability would depend on how "optimally" eduction is decentralized among the multiple masters for each program.

6 Conclusions

GLU is a hybrid intensional language for programming conventional parallel computers. It consists of Lucid, a purely intensional language for describing parallel program composition using imperative functions expressed in sequential languages. Parallelism in GLU programs is discovered at runtime using eduction, a demand-driven model of computation. The challenge for compiling GLU programs to conventional parallel computers is to employ those eduction-based implementation models that can efficiently and scalably discover and exploit inherent parallelism.

We have shown that the stateless worker variation of the single-master implementation suffers from the master being the bottleneck especially for small granularity of parallelism and large number of workers. We have also shown how making the workers stateful, even at the expense of static mapping of imperative functions to workers, has a positive effect on performance scalability. We have proposed a new implementation for GLU programs, based on multiple masters, that reflects a fully decentralized embodiment of the eduction model. We postulate that this implementation can potentially offer superior performance scalability when compared to the single-master implementations even when the granularity of parallelism is not very coarse.

Immediate future work will focus on evaluating the performance scalability of the new implementation using GLU applications with modest granularity of parallelism. In the long term, we will focus on ways in which the new implementation can be used for heterogeneous parallel systems and to tolerate single-point failures in such systems as well as automatic generation of mapping functions for GLU programs through compile-time analysis.

References

[1] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In L. Bic, J-L. Gaudiot, and G. Gao, editors, *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, 1995.

[2] R. Jagannathan and C. Dodd. GLU programmer's guide (version 0.9). Technical Report SRI-CSL-94-06, Computer Science Laboratory, SRI International, Menlo Park, California 94025, U.S.A, July 1994.

[3] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional Programming*. Oxford University Press, New York, 1995. ISBN 0-19-507597-8.

[4] 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).

[5] K.M. Chandy and S. Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett Publishers, Boston, 1992.

[6] D. Gelernter and N. Carriero. Coordination languages and their significance. *CACM*, 35(2):97–107, February 1992.

[7] W-C Du. An intensional approach to parallel programming. *IEEE Parallel and Distributed Technology*, 1(3):22–32, August 1993.

[8] A.A. Faustini and W.W. Wadge. Intensional programming. In J.C. Boudreaux, B.W. Hamill, and R. Jernigan, editors, *The Role of Languages in Problem Solving 2*. Elsevier Science Publishers B.V. (North-Holland), 1987.

[9] E. Moreno, A. Fafitis, and A.A. Faustini. Parallel execution for the finite element analysis of box girder bridges. In *Proceedings Fourth International Conference on the Applications of High Performance Computing in Engineering*, Milan, Italy, June 1995. (to appear).

[10] I. Agi. Glu for multidimensional signal processing. In *Proceedings of ISLIP'95: The Eighth International Symposium on Languages for Intensional Programming*. World-Scientific, 1995.

[11] I. Agi and R. Jagannathan. A portable fault-tolerant parallel software mpeg-1 encoder. In *Proceedings Second IASTED/ISMM International Conference on DIstributed Multimedia Systems and Applications*. IASTED/ISMM, August 1995.

[12] R. Jagannathan and C. Dodd. Application experience with an implicitly parallel composition language. In *Proceedings 1994 USENIX Very High-Level Languages Symposium*. USENIX Association, Berkeley, California, October 1994.