GLU: A High-Level System for Granular Data-Parallel Programming*

R. Jagannathan
C. Dodd
I. Agi
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, California 94025

email: {jagan,dodd,agi}@csl.sri.com
www: http://www.csl.sri.com

October 27, 1995

*Final draft of paper to appear in **Concurrency: Practice and Experience (John Wiley)**

Abstract

We describe a high-level system for granular data-parallel programming called GLU in which parallel applications are described as succinct implicitly parallel intensional compositions using sequential imperative functions. We show how different architecture-independent parallel programs can be generated from the high-level application description. We also show how GLU enables functional debugging of parallel applications without requiring their parallel execution. To illustrate the efficiency of parallel programs generated in GLU, we consider the results of a performance study of three real parallel GLU applications, executed on two different parallel computers. Finally, we compare GLU to other very high-level systems for granular data-parallel programming.

1 Introduction

It is well known that applications in scientific computing are rich in a particular kind of parallelism, namely data parallelism. It is also well known that by harnessing data parallelism, not only can important scientific problems be solved faster but it would be possible to deal with hitherto intractable problems (commonly referred to as "grand challenges"). Data parallelism can occur at different levels of granularity — from fine grain (SIMD) to coarse grain (SPMD). Although many of the early parallel computers were designed to exploit fine-grain data parallelism, the focus of more recent parallel computers has shifted to effectively harnessing coarse-grain data parallelism [7]. Although parallel computers that can exploit massive amounts of data parallelism have been built, being able to rapidly develop portable and efficient parallel applications for such computers remains a significant impediment.

Existing systems for data-parallel programming can be broadly divided into two categories — low-level and high-level systems where "level" refers to the level of transparency at which parallelism is expressed and discovered. A typical low-level system would require the application to explicitly specify all interprocess communication and synchronizations using message-passing calls provided by the system. On the other hand, a typical high-level system would require the application to specify the manner in which different parts of the application are related in terms of dependencies or constraints. Examples of low-level parallel programming systems are PVM [23, 12], MPI [13], and Linda [8]. Examples of high-level languages are PCN [9], CC++ [24], HPF [18], HeNCE [6], and CODE [21].

Whereas low-level systems give the application direct control over how parallelism is to be managed, they tend to be difficult to use and maintain. In some cases, they compromise the degree to which applications can be architecture independent. On the other hand, high-level systems are easy to use and offer architecture independence. However, applications developed in these systems tend to be less efficient than their low-level equivalents. Notwithstanding the many advantages of the high-level approach to parallel programming, it is important that a high-level system exhibit comparable execution efficiency when compared to low-level systems.

In this paper, we describe a high-level system called GLU for developing scientific and engineering data-parallel applications that has been designed with the following goals in mind.

- Ease of use
- Architecture independence
- Support for existing sequential code
- Efficiency across multiple architectures

GLU is based on a hybrid language comprising an intensional language (Lucid) for specifying the parallel *structure* of the application and an imperative language (C) for specifying the various application functions. The intensional part of a typical GLU application is quite small when compared to the procedural part of the application. The procedural part typically corresponds to code fragments from existing sequential applications that are suitably *wrapped.* Since parallelism is *implicit* in the intensional specification, the GLU system can tune the application executables to take advantage of the kind and granularity of parallelism as well as the target architecture and its OS-level features.

The rest of the paper is organized as follows. We illustrate the expressivity of the GLU language by showing how a prototypical data-parallel application can be constructed by reusing existing sequential code. We then describe how different parallel programs can be generated from the high-level application description and compiled to different parallel architectures. We also consider how parallel GLU applications can be functionally debugged without requiring parallel execution. We describe the performance of three real GLU applications on two different parallel machines — CM-5 and NOW (network of workstations). Last, we compare GLU to other high-level systems for granular parallel programming.

2 Expressing Data Parallelism in GLU

We illustrate here, the expressivity of GLU in different kinds of granular data parallelism. We then show how an existing imperative application can be transformed to a parallel GLU application with only nominal modification.

2.1 Lucid – The Intensional Core of GLU

Lucid [4] is the multidimensional intensional language used by GLU to express the parallel structure of an application. The language Lucid is based on intensional logic, a form of mathematical logic commonly used in natural language semantics. In an intensional language, meanings of expressions do not just depend on the constituents of the expression — they also depend on a set of underlying implicit *contexts* (or dimensions). This property, which we call intensionality, makes it possible for multidimensional computations to be naturally and succinctly expressed in Lucid. Latent in such Lucid programs are massive amounts of parallelism, notably data parallelism.

Consider a Lucid program to compute the dot product of two vectors, `a` and `b` varying in dimensions `i` and `j`, respectively, producing a vector `c` varying in dimension `k`.

```
c where
    c = ( realign.i,k a ) * ( realign.j,k b )
  end
```

This program defines `c` is the result of pointwise multiplication of two vectors in the `k` dimension, where the two vectors are the dimensionally realigned versions of vectors `a` and `b`. What this program does not express, which is implicit, is that each element of `c` can be independently computed by multiplying the respective elements of `a` and `b`. This is a simple example of data parallelism that is latent in a Lucid program.

Now consider a Lucid program to compute the transposition of a matrix `a` in dimensions `i` and `j` in matrix `b`.

```
b where
    b = realign.t,i ( realign.i,j ( realign.j,t a ) );
  end
```

This program defines `b` as a series of `realign` applications on `a`. The operator `realign` realigns an object that varied in one dimension to vary in another another dimension. Figure

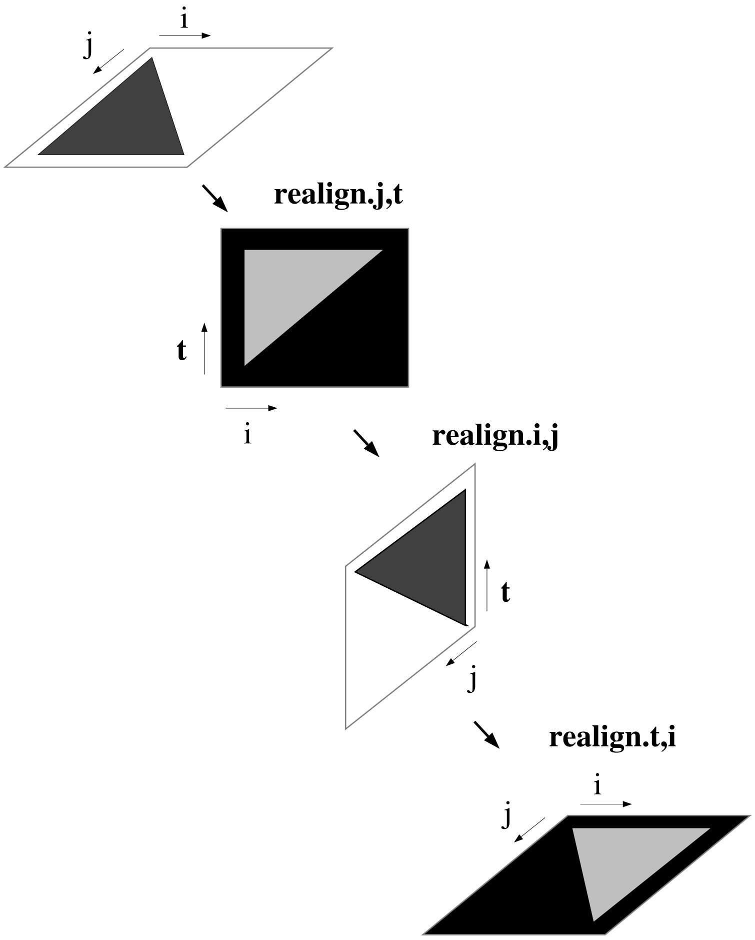

Figure 1: Lucid-style Matrix Transposition

1 illustrates the staged approach using realign to matrix transposition. With the innermost realign, a is realigned so that what varied in the j dimension varies in the t dimension, a new temporary dimension. With the enclosing realign, what varied in the i dimension varies in the j dimension. And with the outermost realign, what varied in the t dimension varies in the i dimension. The effect is that the i and j dimensions of a are switched in b.

Implicit in the matrix transposition is considerable data parallelism. Although for a given element of a, the application of the three realign operations has to proceed sequentially, this can occur simultaneously with identical applications of the sequence of realign operations for each of the elements of a.

Here is a Lucid program to add the first n elements of a vector a varying in dimension i.

```
s asa.t #.t == n where
  dimension t;
  s = 0 fby.t s + (a @.i #.t);
end
```

This program introduces two new Lucid operators — asa (as soon as) and fby (followed by). It also introduces a new dimension t using the dimension declaration. The head of the where clause, s asa.t #.t == n has the value of s at point n in the t dimension. The definition of s says that its value at point 0 in the t dimension is 0, and at each succeeding point its value is the sum of the value of s at the previous point and the value of a at the previous point (in the i dimension since a varies in the i dimension.)

Informally, the value of s at each point T in the t dimension is the sum of values of the first T-1 values of a. Thus, the first value of s is 0, the second value of s is the first value of a, the third value of s is the sum of the first two values of a, and so on. The program returns the sum of the first n values of a.

The above program is sequential since the values of a are accumulated in s in a serial manner. Since addition is associative, it is possible to compute the sum in parallel by adding the first n elements of a pairwise, adding the n/2 results pairwise, until only one result remains, which is the desired sum. The Lucid program to do the parallel summation is as follows.

```
linear_tree.i( add, a, n ) where
  add( x, y ) = x + y;
end
```

Function linear_tree is a Lucid function for tree computations that is used to perform binary summation of the first n elements of a. It has considerable data parallelism, as adding pairs of numbers at each stage of the summation tree can be conducted in parallel.

We can now define a program to multiply two matrices, a and b, varying in dimensions i and j.

```
linear_tree.k( add, (realign.j,k a)*(realign.i,k b), n ) where
  dimension k;
  add( x, y ) = x + y;
end
```

This program uses vector multiplication described earlier to multiply the row i of a with column j of b. It uses binary summation introduced earlier to sum the inner product. It has significant amounts of implicit data parallelism — each element of the result matrix can be computed in parallel, and computation of each element requires computing the sum of the product of two vectors, both of which are in themselves highly data parallel.

2.2 Granulating Lucid

Despite the attractive properties of Lucid, it has two disadvantages. First, it is difficult to effectively implement Lucid on conventional parallel machines, which are not well suited for exploiting fine-grain data parallelism. Second, it would require existing sequential applications that are written in imperative languages to be completely rewritten in Lucid — possibly a Herculean undertaking.

To overcome these two disadvantages of `Lucid`, we have developed the hybrid language with the same name as the system, namely, `GLU` (which stands for Granular LUCID). A `GLU` program consists of a `Lucid` part and a `C` part. The `Lucid` part specifies the program in terms of data dependencies between user-defined `C` functions. These `C` functions can be developed from code fragments of existing sequential applications. With `GLU`, the granularity of parallelism, being coarse, is readily amenable to efficient exploitation. Also, GLU programs can reuse code from existing sequential applications by way of C functions.

We illustrate the hybrid nature of the `GLU` language by considering matrix multiplication again. The `GLU` program is given below.

```
linear_tree.k( add,
               mult( (realign.j,k a), (realign.i,k b) ),
               n / blocksize )
where
  dimension k;
  matrix add( matrix, matrix );
  matrix mult( matrix, matrix );
end
```

The hybrid nature of the `GLU` language is reflected in the manner in which the processing is partitioned between the Lucid part and the C part. The Lucid part only specifies the parallel structure of the matrix multiplication algorithm — a tree whose base consists of block matrix multiplications and whose interior consists of block matrix additions. The C part (user-defined functions `mult` and `add`) specify sequential matrix multiplication and sequential matrix addition, respectively.

The main difference between the `Lucid` and `GLU` programs is in the granularity of parallelism they express. Parallelism in `Lucid` programs is at the level of the basic operations (such as `*` and `+`), and thus fine grain. Parallelism in `GLU` programs is coarser grain — it is at the level of imperatively specified functions, namely `mult` and `add`. Observe that apart from this difference, the two programs are identical.

2.3 A Granular Data-Parallel Application

Raytracing is a widely used graphics application that exhibits a high degree of granular data parallelism. The basic idea in raytracing is to render a two-dimensional (2-D) view of an observer of a three-dimensional (3-D) scene consisting of objects with different reflective and refractive properties and multiple sources of light. The 2-D view is a pixel plane where the color (RGB value) of each pixel is determined by the raytracing algorithm. The standard algorithm is such that the RGB value of each pixel can be computed independently.

The sequential implementation of raytracing that we started with consisted of approximately 3000 lines of C code over 25 files. The `main` function of the implementation processes the various command line options, initializes global data structures, reads the scene data, and invokes the function `Screen` to perform raytracing. This function calls one of three slightly different functions, each of which essentially consists of a doubly nested loop that computes the RGB value of each pixel in the 2-D image using function `Trace`.

The principal source of parallelism in this application is in the unraveling of the doubly nested loop such that each iteration can essentially proceed independently. However, invoking each iteration independently has associated overhead that can obviate the benefits of doing so. Therefore, it is necessary to granulate the parallelism by making each iteration serially compute RGB values of several pixels while reducing the number of iterations. Although this conceptually reduces parallelism, it reduces the overhead per pixel, which improves the effectiveness with which parallelism is exploited.

In converting this application to GLU, we expose the parallel structure of raytracing in the composition part while subsuming the per-pixel processing in sequential functions. The GLU program composition for the raytracing application is

```
struct INPUT
{
  int x_dimension;
  int y_dimension;
  int grainsize;
  char name[64];
};
typedef struct INPUT INPUT;
struct STRIP
{
  int cols, rows;
  unsigned char line[cols][rows][3];
};
typedef struct STRIP STRIP;
local int display( string, int, int, int, int, int, int, STRIP *);
      STRIP *raytrace( int, int, int, int, int, int, int, int, int );
local INPUT *input( int );
local int xdim_in( INPUT *);
local int ydim_in( INPUT *);
local int grainsize_in( INPUT *);
local string appl_name_in( INPUT * );
local int reset( int );

pics where
  pics = pic;
  pic = image( xdim, ydim, ntiles );
  image( xd, yd, n ) = reset( p )
    where
```

```
          p = planar_tree.x,y(
                combine,
                display( appl_name, x, y, xdim, ydim, ifactor, ifactor, tile )
                  where
                    tile = raytrace( time, x, y, xdim, ydim,
                                     ifactor, ifactor, xdim, ydim );
                    ifactor = sqrt( n );
                  end,
                 n )
              where
                dimension x, y;
                combine( a, b, c, d ) = a + b + c + d;
              end;
      end;
    ntiles = xdim * ydim / grainsize;
    xdim = xdim_in( inp );
    ydim = ydim_in( inp );
    grainsize = grainsize_in( inp );
    appl_name = appl_name_in( inp );
    inp = input( #.time ) fby pic,inp;
  end
```

We introduce two new type definitions: `INPUT` and `STRIP`. These are used by sequential functions that provide the interface between composition and existing sequential code. We define a function `input` that reads application parameters from `stdin`. Thus, the command-line interface to the application has changed. Four simple functions — `xdim_in`, `ydim_in`, `grainsize_in`, and `appl_name_in` — are defined, each of which extracts the appropriate field from the structure returned by `input`. We define function `display`, which when invoked with several arguments including an image, superimposes the argument image on an existing X window. Associated with `display` is function `reset` that creates a new X window to display images. Finally, we define function `raytrace`, which accepts nine integer arguments that help define the set of pixels of the image that it is responsible for computing using existing functions such as `Trace`. Also `raytrace` is the only function that is executed remotely; all other functions, being `local`, are executed locally, as we describe later.

The GLU program defines a stream of images (in `pics`) where each image is denoted by the variable `pic`. This variable is defined as the GLU function `image` applied to `xdim`, `ydim`, and `ntiles` where these arguments are provided as, or derived from, `inp` (returned by `input`). GLU function `image` returns the vertex of a 2-D pyramid computation (specified by `p` shown in Figure 2).

Each invocation of `display( raytrace( ... ) )` at the base of the pyramid computes and displays a set of pixels independent of any other invocation. The number of such invocations, and hence the degree of parallelism, is given by `ntiles`. The interior of the pyramid simply computes the sum of the values returned by the base invocations, although the result is not of any use to the application user. When the vertex of the pyramid has been computed, function `reset` is invoked, which creates a new X window to display images.

The new sequential code that we have generated corresponds to functions `input`, `xdim_in`, `ydim_in`, `grainsize_in`, `appl_name_in`, and `reset`, which together amount to less than 40 lines of C code and function `raytrace`, which is mostly a rearrangement of existing code (from

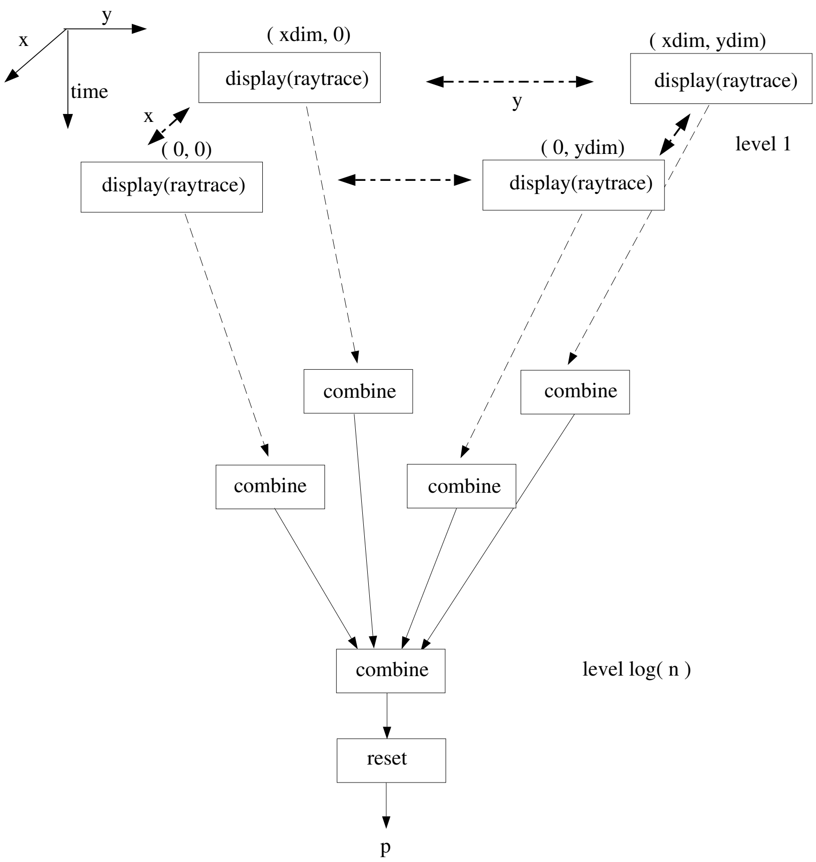

Figure 2: 2-D Raytracing Computational Pyramid

functions `main` and `Screen`) with around 10 to 20 lines of new C code. The parallelization of the sequentially written raytracing application into GLU required around 2 to 3% new C code and approximately 60 lines of the GLU composition.

3 Parallel Program Generation

We have shown how parallel applications can be described in GLU at a high level of abstraction where both expression and exploitation of parallelism are implicit. Abstractly, a GLU application can be evaluated using a parallel model of computation called eduction [3, 4]. For each high-level application description, the GLU system generates a parallel program that when executed on the target system will eductively discover and harness program-inherent parallelism. The generated parallel program, which reflects a particular embodiment of eduction, is referred to as the parallel program architecture (PPA). The GLU system currently supports three such architectures — generator/dynamic-worker, generator/static-worker, and distributed-generator.

3.1 Eduction Model of Computation

Eduction is a demand-driven model of computation. The basic idea in eduction is to compute a value only when it is *needed* even though it could have been computed earlier (which is the case with data-driven computation). Associated with each basic operation are rules for demand propagation — for strict operations, both arguments are demanded at the appropriate contexts and for nonstrict operations arguments are demanded in the necessary order. Procedural functions are treated as strict operations. User-defined Lucid functions are evaluated *by need* — demand for the result of a Lucid function at some context results in demand for the body of the Lucid function at the appropriate context, which eventually results in function arguments being demanded as and when needed at appropriate contexts.

The eductive evaluation of the vector dot product program given in Section 2.1 proceeds as follows. Demand for `c` at context `k=K` results in two simultaneous demands — `realign.i,k a` at context `k=K` and `realign.j,k b` at context `k=K`, each of which results in demands for `a` at context `i=K` and `b` at context `j=K`. Assuming that these demands can be immediately satisfied, their product can be computed, which will satisfy the original demand for `c` at context `k=K`. Observe that, in principle, demands for different values of `c` can be independently and simultaneously propagated and satisfied.

Now consider eductive evaluation of the matrix transposition example given in Section 2.1. Demand for `b` at context `i=I,j=J` results in demand for `realign.t,i(realign.i,j(realign.j,t a))` at context `i=I,j=J,t=0`. This results in demand for `realign.i,j( realign.j,t a )` at context `i=I,j=J,t=I`, which in turn results in demand for `realign.j,t a` at context `i=J,j=J,t=I`. This causes demand for `a` at context `i=J,j=I,t=I`, which is equivalent to demand for `a` at context `i=J,j=I` (since `a` and `b` do not vary in the local `t` dimension). Again observe that demands for different values of `b` can propagate and be satisfied simultaneously.

3.2 Generator/Dynamic-Worker PPA

The generator/dynamic-worker PPA consists of a single generator and multiple workers. (It is similar to the "crowd computation" model for parallel programming [12].) The generator is responsible for all aspects of eductive evaluation except for execution of certain imperative user-defined functions. This is done at the workers 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 workers 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 a worker relative to the time taken to execute the ripe functions at the worker.

Load balancing is dynamic. Each worker is capable of executing any imperative function and no state is kept at any executor such that the result of executing an imperative function depends on that state. Each worker waits for the generator to provide ripe functions (i.e., functions whose arguments have been computed). When a worker is given a ripe function to execute, it unblocks, executes the function, returns the result to the worker, and blocks again.

3.2.1 Generator/Static-Worker PPA

An important property of the generator/dynamic-worker PPA is that workers are considered to be memoryless — correct execution of a ripe function at a worker is independent of the history of ripe function executions at that worker. This memoryless-ness makes dynamic load balancing possible, as a ripe function can be executed at any worker. It also makes dynamic scalability possible, as workers themselves can be added and removed at will during program execution. Memoryless-ness also means that ripe functions must be self-contained — both the name and all the arguments of the function must be transported from the generator to a worker for successful execution. The drawback of doing so is that the cost of transportation of argument data, all of which is stored at the generator, may be a significant fraction of the amount of time spent in executing the ripe function at the worker, thus reducing the benefits of remote execution.

For programs where the granularity of parallelism (ratio of execution time to communication time) is not sufficiently coarse, it is possible to reduce the overhead of data transportation by storing the data in each of the workers (instead of all at the generator) and executing specific functions on specific workers. The decomposition of data across workers is specified in the GLU program itself using "mapping expressions" associated with each variable definition. At runtime, each ripe function is executed on the most suitable worker from a data proximity standpoint. Data needed by a ripe function executing on a worker is stored in another worker. It is retrieved directly from the other worker transparent to the program.

While the generator/static-worker PPA reduces the amount of data communication overhead, it requires load balancing to be more statically determined. It also complicates the extent to which workers can be dynamically added or removed when compared with the generator/dynamic-worker PPA.

3.2.2 Distributed-Generator PPA

A limitation of both the generator/worker PPAs we have described is that the generator cannot support more than a certain number of workers without becoming a bottleneck.

Another limitation is from a fault-tolerance standpoint — if the generator fails, then the entire application would also fail.

The distributed-generator PPA consists of multiple peer processes, each of which acts as both a generator and a worker. That is, each peer is responsible for eductive evaluation of certain parts of the program as well as execution of imperative functions. In addition, each peer can take over responsibility of another peer that fails.

The division of eductive evaluation across peers is determined by mapping expressions associated with each variable definition. At runtime, demands for values are either processed internally at a peer where they are generated or passed to the appropriate peer for further processing. When the value is eventually produced, the demand is passed back to the original peer.

The two advantages of the distributed-generator PPA are scalability in that peers can be added without creating a bottleneck at any one process, and fault tolerance in that no single-process failure will cause the application to cease operation.

3.3 Parallel Program Compilation

The parallel program generated by the GLU system is essentially architecture independent in the sense that it can be compiled for different processor and operating systems combinations and can use different interprocess communication and synchronization mechanisms. Figure 3 illustrates the parallel program compilation process.

The parallel program generation stage of the GLU compiler is responsible for generating C code from a GLU program, which consists of a Lucid part and a C part. The C code generated from the Lucid part of the GLU program is in three files — `-gen.c`, `-wrk.c`, and `-com.c`. The file `-gen.c` consists of code that describes the generator associated with the GLU program whereas `-wrk.c` consists of code that describes the worker associated with the GLU program. The file `-com.c` consists of code that is in common between the generator and worker.

The generator executable is derived by compiling the `-gen.c` and `-com.c` files along with the C part of the GLU program and using suitable runtime libraries. Note that the generator executable can be used as the generator in generator/worker PPA as well as the peer in the distributed-generator PPA if given suitable runtime options.

The worker executable is derived by compiling the `-wrk.c` and `-com.c` files along with the C part of the GLU program and using suitable runtime libraries.

The runtime libraries contain all the architecture-dependent features, thus preserving architecture independence until the executables are finally produced. For example, it is possible to generate executables for entirely different hardware and operating systems from the same C code generated in the parallel program generation phase. It is also possible to employ entirely different communication and synchronization mechanisms at this stage.

Interaction between multiple processes of the parallel program is defined in terms of a high-level asynchronous remote procedure call (ARPC) interface that is independent of particular communication and synchronization mechanisms. When a process (client) invokes a remote procedure call on another process (server), the client is blocked until the server returns. The server can, however, return a "handle" in lieu of the return result even before it executes the invoked procedure, thus unblocking the client to continue possibly invoking other remote procedures. The client, when notified at a later time by the server that the

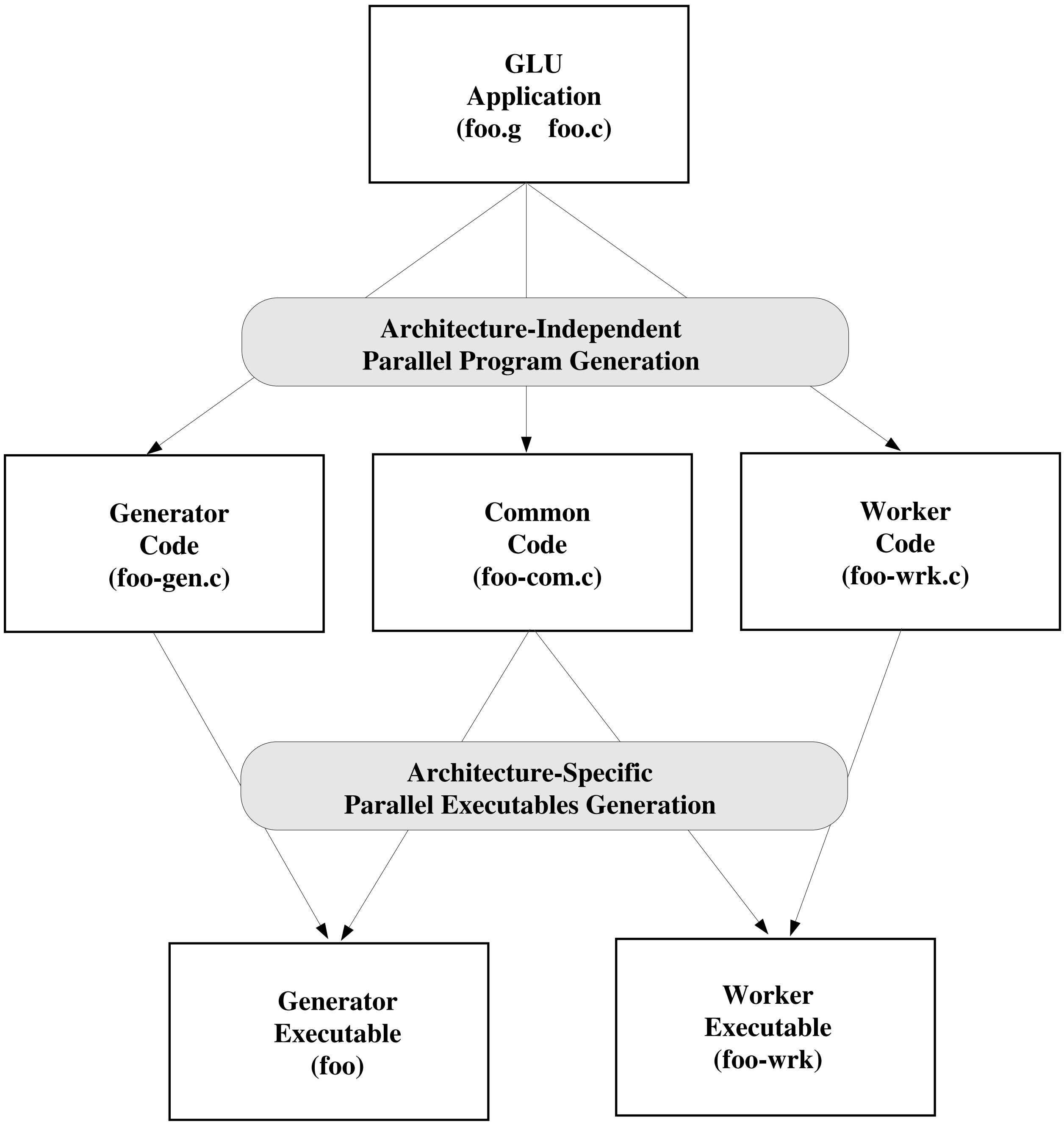

Figure 3: Parallel Program Compilation

result is indeed available, can use the handle to retrieve the result returned by the server.

Interprocess interaction can be distinguished in two ways. The first distinction is between whether memory is shared between processes or is local to each process. In the former case only addresses of data need to be exchanged, whereas in the latter case data itself needs to be transported. The second, orthogonal, distinction is the manner in which information (address or data) is transported between processes. Currently, our ARPC supports several transports, including TCP/IP, PVM, and CMMD, and we are working on other transports as well such as MPI and AAL5/UNI-3.0 (ATM).

4 Debugging

Two important facets of debugging parallel programs are functional debugging and performance debugging. Functional debugging is concerned with whether a program produces the correct results, whereas performance debugging is concerned with the efficiency with which the correct results are produced.

4.1 Functional Debugging

The parallel program generated by the GLU system is sequentially debugged to ensure correct functionality. Sequential functional debugging of a GLU program consists of two aspects: Are the data dependencies expressed in the Lucid part of the GLU program correct? Does each C function produce the correct results?

Data dependencies in a GLU program are examined at runtime using an *eduction trace*. This trace reveals the propagation of demands and their subsequent satisfaction through production of values. Using this trace, it is possible to verify the data dependencies specified by the program. The trace also reveals the values produced as a first-level examination of C function behavior.

The C functions of a GLU program are debugged in a conventional manner (using a conventional debugger). Such debugging is initiated when a C function is invoked for a given set of arguments.

The functionality of a GLU application (i.e., the results it produces for a given set of inputs) is independent of how it is implemented. That is, both a sequential execution and a parallel execution of the parallel program corresponding to the GLU application would yield the same results. Therefore, it is possible to develop and functionally debug a GLU application in an entirely sequential mode (e.g., on a workstation) and expect the functionality to be the same regardless of its mode of execution.

Unlike debugging required with low-level programs, there is no need to be concerned with race conditions, detailed knowledge of states of interacting processes, deadlock, and the effects of debugging on program functionality.

4.2 Performance Debugging

Unlike functional debugging, performance debugging for GLU programs is at the level of the PPA because sources of inefficiencies need to be understood at that level. When GLU programs are compiled using the centralized generator model, both the generator and executor have performance measurement instrumentation built in. When the program is executed with performance logging turned on, the states and activities of the generator and executors are recorded in a logfile. From this logfile, various measures of program performance can be extracted after execution. For each executor, it shows the average utilization, first and second moments for remote function execution time and grain size, and the average time taken to get the arguments and return the result. For the generator, it shows the total runtime, idle time, local function execution time, communication time, and eduction time. The summary shows the average generator and executor utilizations, average grain size, and communication overhead as a fraction of grain size.

The main objective of performance debugging is to increase the average executor utilization since that is a very good measure of parallelism being exploited. This can be achieved by reducing communication cost (improving granularity) and by reducing generator utilization (reducing overhead and unnecessary sequentiality).

5 Performance of GLU Applications

We consider the performance of three large GLU applications derived from their sequential counterparts on two different parallel systems – NOW and the CM-5 MPP.

The three applications we study are raytracing [15], video compression [2], and computed tomography (CT) reconstruction [1], each of which is computationally intensive and exhibits a considerable degree of granular data parallelism. Here, we describe the parallelization of the video compression and CT reconstruction applications. (We described the parallelization of raytracing earlier.)

5.1 MPEG-1 Video Encoder

The MPEG encoder compresses a color video sequence using a combination of compression schemes, and produces a serial bit stream that can be sent over a communications channel. At the end of this channel, a MPEG decoder processes this bit stream and reproduces a facsimile of the original video sequence. The MPEG encoder is by far the most computationally intensive part of this scheme.

MPEG encoding of a video sequence requires many compression steps. Each frame in the video sequence is compressed using a combination of block-based motion compensation techniques to take advantage of interframe temporal redundancy, and discrete cosine transform (DCT)-based compression to take advantage of intraframe spatial redundancy [16]. Each frame (picture) in the sequence is broken into 8 x 8 pixel *blocks* for interframe DCT compression and 16 x 16 pixel blocks for intraframe motion compensation.

A number of these blocks strung together form a *slice* (of a frame) that acts as a resynchronization unit. These slices combine to form a *picture*. Pictures are grouped to form a random access unit so that the video can be viewed either forward or backward. This group of pictures (GOP) has little or no dependence on the pictures in adjacent GOPs.

The compression using spatial and motion techniques yields a bit stream that is compressed even further by the encoder using a variable-length (Huffman) coding. This resulting bit stream becomes the MPEG sequence with the addition of the necessary sequence and GOP headers.

In the parallel implementation using GLU, a C function calculates and writes a GOP bit stream to a file. This function can execute in parallel, so that multiple GOP bit streams can be generated at the same time and written to files. The generated GOP bit streams are concatenated in serial order to form the MPEG bit stream.

The GLU function GenMPEGStream is used to implement the MPEG encoder:

```
GenMPEGStream(numFrames, ofp, outputFileName, globals, huffTable)
  = linear_tree.gop( same, N, numGOPs+1 ),GOPStoMPEG(GOPNumber, GOPLength, N)
    where
     index gop;
     numGOPs = numFrames / framePatternLength;
     framePatternLength = GetPatternLength(globals);
     N = 0 fby.gop GOPStoMPEG(GOPNumber, GOPLength,  N, outputFileName);
     frameNumber = 0 fby.gop frameNumber + framePatternLength;
     GOPLength = processGOP(InputFileNames, GOPOutputFileName,
            ofp, GOPNumber, globals, huffTable);
     InputFileNames = GetInputFileNames(frameNumber, globals);
     GOPOutputFileName = GetGOPOutputFileName(GOPNumber, outputFileName, globals);
     GOPNumber = GetGOPNumber(frameNumber, globals);
     same(a,b) = a+b-a;
end;
```

This implementation of GenMPEGStream forces the ordering of the GOPs sent to GOPStoMPEG to be sequential. The generation of the GOPs takes place in the function processGOP. The main source of granular data parallelism is in being able to have several instances of processGOP active at the same time.

The MPEG encoder can be parallelized at many levels, from the pixel level up to the GOP level. Our implementation breaks up the calculation at the level of the GOP mainly because individual GOPs can be processed independently and also because existing sequential code generates the MPEG bit stream on a per-GOP basis.

5.2 Computed Tomography Reconstruction

Computed tomography is another application that requires substantial computing power. X-ray attenuation data measured by a set of sensors in a CT scanner form the radon transform of a slice through a 3-D object. The radon transform of the object can be thought of as the set of shadows of the object, illuminated over π radians by X-radiation. The strips of the set of shadows defined by the plane that intersects the Radon transform are called the *projections* of the slice (or tomograph) through the object. The inverse Radon transform, commonly implemented as a *filtered backprojection*, reconstructs a tomograph from the CT projection data [11]. Each tomograph of the object can be reconstructed independently. A set of tomographs reconstructed perpendicular to the plane of intersection generates a 3-D data set that represents the average volumetric density throughout the object. A useful property of the filtered-backprojection reconstruction scheme is that the processing of each projection is independent of all other projections, thus simplifying parallel computation.

CT reconstruction from large data sets using filtered-projection techniques takes on the order of minutes to complete on a single processor computer. Reconstruction of large or dense objects often requires iterative techniques that map the data back and forth between image space and Radon space, requiring the implementation of the forward Radon transform as well [19, 5]. If 3-D reconstruction is required, the computation time can easily take hours to complete. Several specialized hardware architectures have been proposed to reduce the reconstruction time [22, 14]. We use a software approach.

The CT code performs three functions—projection filtering, backprojection, and the forward Radon transform. These three operations can be used in a variety of ways. Single-pass reconstruction requires filtering and backprojection. Iterative reconstruction schemes require all three operations. In our case, the forward transform is used to calculate the projections from a test image, so that various reconstruction algorithms and finite-precision effects can be simulated.

For single-pass reconstruction the projections are saved as an image in polar coordinates. Each invocation to compute the projections (function `ProcessProjections`) is assigned a unique subset of the projections, as calculation of each projection is independent of the other.

Each invocation of function `ProcessProjections` generates a partially reconstructed image by backprojecting its set of filtered projections. Since the backprojection is associative, the partially reconstructed images can be added together to fully reconstruct the image. This addition operation is performed by function `AddImage`.

The GLU function `GluRadon` used to implement CT reconstruction is

```
GluRadon ( imageIn, globals, theta0, N)
   = linear_tree.c ( AddImage, ProcessProjections(imageIn, globals, theta), N)
where
   dimension c;
   theta = theta0 fby.c theta + (M_PI/N);
end;
```

The function `ProcessProjections` takes the input image, the global constants, and the starting angle as inputs, and returns a partially reconstructed image. N of these partial reconstructed images are produced and passed to `AddImage` to be combined into a single image. The starting angle for each worker is incremented by π/N until all the angles are covered.

5.3 Experiments and Results

Performance of each of the three applications was studied on a NOW platform consisting of as many as 16 essentially unloaded sixteen Sun Sparcstation 2 (SS2) workstations[1] connected by a 10 Mbps Ethernet network and on a CM-5 consisting of upto 128 processors (where each processor has similar performance to a SS2 workstation).

Each application was first translated to a generator/dynamic-worker PPA and then compiled to target-specific generator and worker executables. Raytracing was applied on a 512×512 image, consisting of 64 32×32 independent tiles.

MPEG-1 video compression was applied on 256 360×288 YUV-format frames with GOP size of 4, resulting in 64 independent GOPs.

CT reconstruction was applied on a 128×128 image in PGM format with 2048 independent projections grouped into 64 sets of 32 projections each.

[1]All product and company names mentioned in this paper are the trademark of their respective holders.

5.3.1 NOW Results

Figure 4 shows the performance speedups for the three applications with an increasing number of workstations (1,2,4,8,16) associated with the workers. The generator runs on a separate workstation. Speedup is calculated by dividing the time taken by the sequential version of the application by the time taken by the parallel GLU version with a specific number of workers, one per processor.

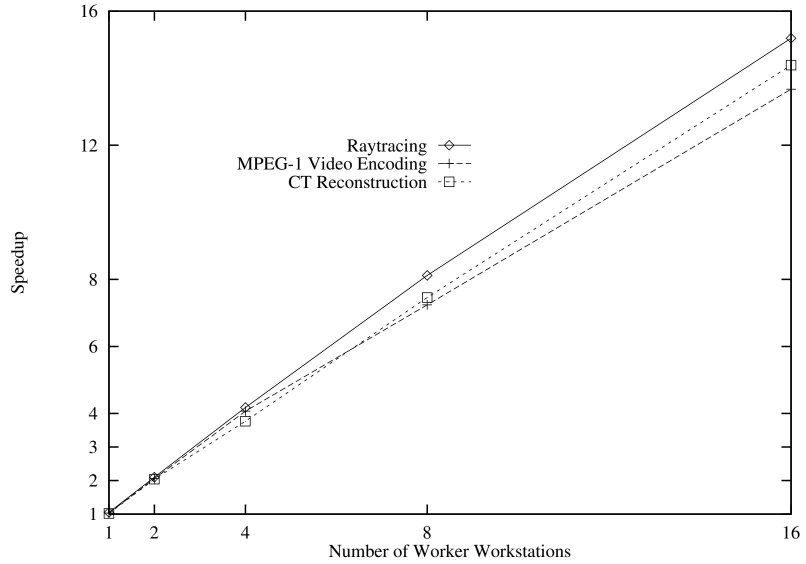

Figure 4: GLU Application Speedups on NOW

Of the three applications, Raytracing shows the most sustained speedup with an increased number of processors whereas MPEG-1 video compression shows the least sustained speedup. While the granularity of parallelism of Raytracing is the smallest (approximately 4 seconds), the amount of communication overhead is only nominal. MPEG-1 video compression, despite having a larger granularity of parallelism (approximately 15 seconds) and very little communication overhead, requires a significant amount of remote file I/O activity. When multiple workers are attempting to perform I/O over Sun's NFS, they tend to get sequentialized, which explains the drop in speedup with 8 and 16 worker workstations. CT reconstruction has the largest granularity of parallelism (approximately 22 seconds) but it also has a moderate amount of communication overhead. This tends to degrade speedup as the number of worker workstations increases.

5.3.2 CM-5 Results

Figure 5 shows the performance speedups for the three applications with an increasing number of processors (1,2,4,8,16,32,64) associated with the workers. The generator runs on a separate processor. Speedup is calculated by dividing the time taken by the sequential version of the application by the time taken by the parallel GLU version with a specific number of workers, one per processor.

The granularities of parallelism for the three applications are slightly higher since a CM-5 processor is slightly slower than a Sun SS2 processor. Raytracing and CT reconstruction show

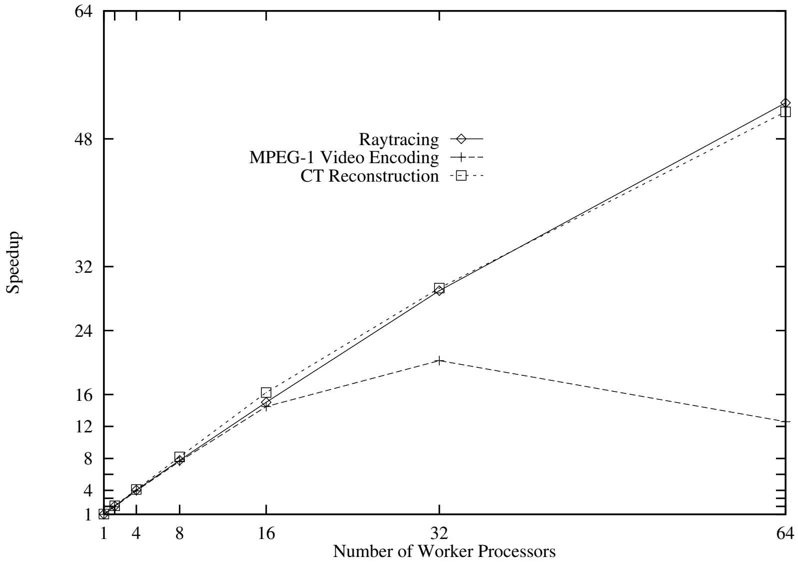

Figure 5: GLU Application Speedups on CM-5

similar speedup characteristics, with CT reconstruction showing slightly better speedup for a smaller number of processors. Note that the communication overhead associated with CT reconstruction is reduced on the CM-5 which has a positive effect on its speedup. MPEG-1 video compression, while exhibiting reasonably scalable performance for 16 processors, starts degrading with 32 processors and deteriorates with 64 processors. This degradation in performance is principally caused by the use of standard file I/O in the application (to read frames and to write compressed bit streams), which when performed simultaneously from an increased number of processors, becomes a significant bottleneck.

6 Related Work

We consider here, the related dataflow-based approaches to programming parallel computers and contrast them with GLU. We also consider other high-level approaches to parallel programming, including PCN, Proteus, HPF, and HPC++.

6.1 Dataflow-based Approaches

HeNCE [6], a graphical parallel programming environment being developed at Oak Ridge National Laboratory, is based on the idea that a parallel program can be represented as a graph whose nodes denote subroutines and whose edges denote data dependencies. HeNCE provides four other graph building constructs — conditional, loop, fan, and pipe. These constructs specify graphs that grow at runtime (through suitable graph rewriting) thus providing a mechanism for specifying dynamic parallelism. Each subroutine in a HeNCE program is compiled with suitable wrapper code as a separate executable, and data dependencies are coded as PVM message-passing calls. A generator program that knows about the program graph manages the execution of the subroutines as separate processes. It provides inputs to the subroutine when invoked and collects the result when execution completes. The scheduling of subroutines to machines is driven by a user-specified cost matrix. While HeNCE and GLU are both based on dataflow graphs, they have several important differences. First, GLU is based on a powerful intensional (multidimensional dataflow) language, Lucid, with which different forms of parallelism can be expressed in a unified manner, whereas HeNCE is based on a simple static dataflow language in which expression of dynamic parallelism requires runtime graph rewriting. Second, the model of computation employed by GLU is eduction that can be implemented using different implementation models, whereas HeNCE loosely corresponds to lazy graph rewriting. Third, the structure of GLU program executables is given by the PPA employed by the compiler, whereas a HeNCE program directly corresponds to the executable architecture — each subroutine is a process and the lifetime of a process is the lifetime of a subroutine invocation.

CODE 2.0 (or simply CODE) [21], a graphical parallel programming environment being developed at the University of Texas (Austin), is based on the idea that a parallel program is a dependency graph of computational units; implicit therein is parallelism. Each node of the graph (which denotes a computational unit) is invoked as per the user-specified firing rules associated with it. While function parallelism is naturally expressed in CODE, data parallelism, the extent of which is usually determined at runtime, requires the use of firing rules as well as arc topology specifications. CODE programs are translated to executables by a series of refinements. While CODE and GLU share the notion of using dataflow graphs to specify the parallel application structure, CODE lacks the expressivity and formal underpinnings of GLU. The model of computation associated with CODE, which is mostly informal, is a form of lazy graph rewriting, similar to HeNCE. While CODE has been implemented on a Sequent multiprocessor and idealized examples have been demonstrated, it does not appear to have been applied to significant applications or to have implementations on different parallel machines.

Cheng and others at Syracuse University (NPAC) have developed a system that uses the AVS framework to build interactive parallel applications from existing parallel programs [10]. The basic idea is to use AVS's dataflow interface for connecting independent parallel (PVM) programs to AVS's visualization framework. The main difference between this approach and GLU is that GLU provides a much richer model of programming than AVS does. Another difference is that AVS composition reflects the architecture of the application, whereas GLU composition can be implemented in different ways using different PPAs.

6.2 Other High-Level Parallel Programming Approaches

PCN [9], a language developed at CalTech, uses a simple yet powerful compositional model with which parallel programs can be constructed by assembling components written in other languages (typically Fortran or C). The compositional model consists of operators (sequential, parallel, and choice composition) for assembling components and shared variables to enable intercomponent interaction. The PCN compositional model requires explicit specification of parallelism (or lack thereof) and explicit use of shared variables to facilitate component interaction. Expressing data parallelism in PCN is accomplished using recursion with the parallel compositional operator. Although parallel implementations of PCN are available, it is unclear as to how efficient these implementations are in prototypical or real-world parallel applications. While PCN and GLU are both hybrid languages, they employ fundamentally different approaches to program composition. PCN uses explicit operators and shared variables, whereas GLU uses multidimensional data dependencies and functions. PCN requires use of recursion to express data parallelism, whereas GLU can express data parallelism using multidimensional data dependencies.

HPF (High Performance Fortran) [18] is a parallel extension of Fortran 90 that supports SPMD programming. An HPF program is a Fortran-90 program with a single thread of control and a global address space. In addition, the programmer annotates the program with data-mapping directives to achieve desired data locality and distribution. HPF programs can be constructed from Fortran 90 although it may be quite difficult to migrate from Fortran 77 without substantial rewriting. HPF supports coarse-grain data parallelism directly, while other forms of parallelism are not easily expressible. HPF programs are architecture independent although it is likely that the data mapping directives may need to be architecture specific to achieve reasonable performance.

HPC++, the emerging C++ equivalent to HPF, is based on a unification of two C++-based parallel languages — CC++ [24] and pC++ [17]. CC++, developed at CalTech, is C++ extended with single-assignment variables, parallel composition and atomic functions. Single-assignment variables enable synchronization between distinct activities. Parallel composition enables multiple C++ statements (which could be function calls) to be invoked simultaneously or multiple iterations of a loop to be invoked simultaneously. Atomic functions enable mutually exclusive execution of a function. Parallelism in CC++ is specified explicitly, and interaction between parallel activities needs to be achieved using single-assignment variables. pC++ enables data-parallel operations using a set of objects of a particular class. It has implementations on several contemporary parallel computing systems.

Proteus [20], a system being developed jointly by the University of North Carolina, Duke University, and Kestrel Institute, is based on a high-level notation that can express both nested data parallelism and process parallelism. Proteus is based on the premise that parallel programming can most effectively be performed by specifying at a high level and using successive refinements to lower levels. In particular, data-parallel Proteus specification can be translated to a lower-level vector library (CVL) that can then be executed on a variety of data-parallel computers. While Proteus is based on explicit imperative specification of parallelism at a high level, GLU is based on implicit declarative specification of parallelism at a high level. Proteus uses successive refinements to generate low-level parallel programs, while GLU uses eduction-based PPAs to generate parallel executables.

7 Conclusions

We have described the GLU system for granular data-parallel programming in which parallel applications are described as intensional compositions of sequential imperative functions. We have shown how an imperatively specified sequential application with inherent parallelism can be rapidly converted to a GLU application with latent granular data parallelism. We have described how the eduction model of computation can exploit this latent parallelism.

We discussed three different eduction-based parallel program architectures to which GLU applications can be translated, and we explained how these generated parallel programs can be compiled to specific target systems. We also discussed how functional debugging of GLU applications can be conducted without parallel execution and we outlined how performance debugging needs to be done at the level of the parallel program architecture.

We considered the performance of three real GLU applications — raytracing, MPEG-1 video compression, and computed tomography reconstruction — on both networks of workstations and the CM-5. We have shown that, for the most part, these applications exhibit very good speedup with increased processing power, and that any significant deterioration in performance is due to reasons unrelated to GLU.

Finally, we compared and contrasted the GLU approach with similar "dataflow-based" approaches and related high-level approaches.

8 Acknowledgments

The work presented in this paper has been supported by an internal grant from SRI International and a grant from the National Science Foundation (Grant # CCR-9203732). The CM-5 experiments as reported in this paper were conducted at both the AHRPC facility associated with the University of Minnesota and the NPAC facility associated with Syracuse University.

We would like to acknowledge the constructive comments provided by the referees on the original manuscript, and we would like to thank Tony Faustini and Ed Ashcroft for their contributions to the GLU system.

For further information on the GLU system, please refer to the following WWW URL – `http://www.csl.sri.com/GLU.html`.

References

[1] I. Agi. GLU for multidimensional signal processing. In *Intensional Programming I*. World-Scientific, 1995.

[2] I. Agi and R. Jagannathan. A portable fault-tolerant parallel software MPEG-1 encoder. In B. Furht and M.H. Hamza, editors, *Proceedings of the Second IASTED/ISMM International Conference on Distributed Multimedia Systems and Applications*, Stanford, CA, August 1995.

[3] 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).

[4] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional Programming*. Oxford University Press, New York, 1995.

[5] S.G. Azevedo. *Model-based Computed Tomography for Nondestructive Evaluation*. PhD thesis, Lawerence Livermore National Laboratories, UCRL-LR-106884, 1991.

[6] A. Beguelin, J. Dongara, G.A. Geist, R. Manchek, and V.S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of SUPERCOMPUTING'91*, November 1991.

[7] G. Bell. Ultracomputers: A teraflop before its time. *CACM*, 35(8):27–47, August 1992.

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

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

[10] G. Cheng, G. Fox, K. Mills, and M. Podgorny. Developing interactive PVM-based parallel programs on distributed computing systems within AVS framework. In *Proceedings of the Third International AVS Conference*, May 1994.

[11] S.R. Deans. *The Radon Transform and Some of its Applications.* Wiley, 1983.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[14] E.B. Hinkle, J.L.C. Sanz, A.K. Jain, and D. Petkovic. P3E: New life for projection-based image processing. *Journal of Parallel and Distributed Computing*, 4(1):45–78, 1987.

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

[16] D. Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[17] J-K Lee and D. Hannon. Object oriented parallel programming: Experiments and results. In *Proceedings of SUPERCOMPUTING'91*, November 1991.

[18] David B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, February 1993.

[19] B.P. Medoff, W.R. Brody, and A. Macovski. The use of a priori information in image reconstruction from limited data. In *Proceedings of ICASSP*, pages 131–134, 1983.

[20] P. Mills, L. Nyland, J. Prins, and J. Reif. Prototyping high-performance parallel computing applications in Proteus. In *Proceedings of 1992 DARPA Software Technology Conference*, pages 433–442. Meridian, Arlington, VA, 1992.

[21] P. Newton and J.C. Browne. The CODE 2.0 graphical parallel programming language. In *Proceedings ACM International Conference on Supercomputing*, July 1992.

[22] E. Shieh, K.W. Current, P.J. Hurst, and I. Agi. High-speed computation of the radon transform and backprojection using an expandable multiprocessor architecture. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(4):347–360, 1992.

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

[24] J. Thornley. Integrating functional and imperative parallel programming: CC++ solutions to the Salishan problems. In *Proceedings of the 1994 International Parallel Processing Symposium*, Cancun, Mexico, 1994.