Application Experience with an Implicitly Parallel Composition Language

R. Jagannathan and Chris Dodd
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, California 94025

{*jaggan,dodd*}*@csl.sri.com*

ABSTRACT

We describe our experiences with a very high-level parallel composition language (called GLU) that enables rapid construction of parallel applications using sequential building blocks (extracted from existing sequential applications) and their execution on diverse parallel computer systems. GLU is sufficiently rich to succinctly express different forms of parallelism — from function parallelism to data parallelism and from pipeline parallelism to tree parallelism. We show by example how a typical sequential application can be converted to a parallel one in GLU and executed on different parallel systems. We also show how GLU has been used to convert two widely used, sequentially written, inherently parallel workstation applications — the **make** utility and a raytracing system — to parallel equivalents that can then be run much faster on a network of workstations.

1 Introduction

The idea of composing new applications from existing ones is commonplace in various Unix[1] shells in which existing filters are often "plumbed" together in a pipeline fashion to form a new composite filter. For example,

```
ps -ax | grep -w "find" | sort | head -1
```

returns the process status description of the alphabetically first process with name `find`. This filter pipeline uses four other existing filters — `ps`, `grep`, `sort`, and `head` — and the plumbing is between the output of `ps` and the input of `grep`, output of `grep` and input of `sort`, and output of `sort` and input of `head`. The filter pipeline execution is basically data driven. However, the amount of data needed to start (and stop) execution varies with each filter — filter `ps` executes first and when it produces its first output filter `grep` also starts executing. Soon after filter `ps` produces its last output and then ceases execution, filter `grep` also stops. Only when filter `grep` produces all its output does filter `sort` perform its function. And as soon as filter `sort` produces its first output, filter `head` starts and completes its execution.

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

Latent in the filter pipeline is parallelism, in which different filters can be executing at the same time. In the above example, filters `ps` and `grep` can be simultaneously active, although filters `sort` and `head` can be active only after both `ps` and `grep` have ceased execution. Such latent parallelism can be exploited when different filters execute on different processors as would be the case when running on a symmetric multiprocessor.

The filter pipeline is an example of an *implicitly parallel* filter composition. Because a filter pipeline is linear and acyclic, it can express only one kind of parallelism — namely, *pipeline* parallelism. The Unix pipeline, which is character oriented, is typically implemented using file I/O, making flow of data through the pipeline much slower.

In this paper, we first review a parallel programming system based on the GLU (pronounced "glue") composition language for constructing parallel application using building blocks of existing sequential code. The language is similar to Unix filter pipelines in its use of dataflow-style plumbing to connect independent functions. We then describe our experiences with constructing parallel GLU applications, from existing sequentially written applications. We show how a prototypical sequentially written application with inherent parallelism can be converted into a GLU application, therein describing the various forms of parallelism that can be expressed using GLU. We then show how GLU can be used to rapidly convert two widely used sequentially written workstation applications — the **make** utility and a raytracing system — into parallel equivalents. We further demonstrate the benefit of doing so by considering application speedup on a network of workstations.

2 The GLU Parallel Programming System

We describe the GLU parallel programming system by first describing the GLU language and then describing how GLU programs are translated to parallel executables. We also briefly consider how GLU compares with related approaches.

2.1 GLU Composition Language

We have developed a composition language [5, 4]), which can be thought of as a generalization of filter pipelines, to enable rapid construction of parallel applications using existing, possibly sequential, functions[2]. The language is sufficiently rich to express different forms of parallelism, including function, data, pipeline, and tree parallelism.

An implicitly parallel composition in GLU consists of two parts: the first part declares the input and output interfaces of sequential functions to be used, and the second part declares the manner in which existing filters are "plumbed" together to form the parallel filter. GLU provides a set of predefined operators that enable arbitrary implicitly parallel composition of filters to be succinctly expressed.

The following illustrates a GLU composition for the above example:

```
string head( string, int );
string sort( string );
string grep( string, string );
string ps_ax();

head( sort( grep( ps_ax(), "find") ), 1 )
```

The filter functions used in the GLU composition are `head`, `sort`, `grep`, and `ps_ax`, whose input and output interfaces are defined in the first part of the program. The second part is a simple linear composition of the filters that corresponds to the Unix filter pipeline described above.

GLU compositions can express more than simply linear compositions. They can express general function compositions with latent structural parallelism. They can also express recursive function

[2] A sequential function in GLU is like a Unix filter in that its output depends only on its inputs.

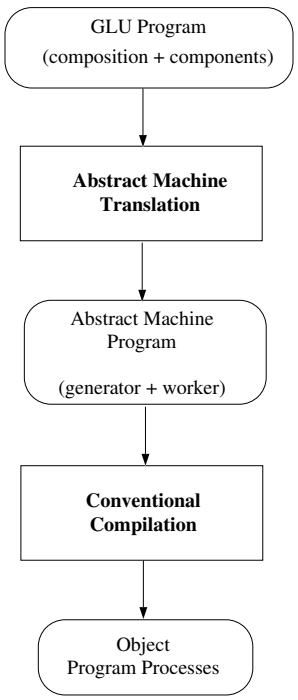

Figure 1: GLU Programming System Architecture

compositions with latent dynamic structural parallelism.

GLU is more than a functional language in which functions can be connected in different ways. GLU is also a multidimensional stream processing language. In the simplest case, a GLU expression denotes an infinite sequence of values in time. For example, the expression `ps_ax()` denotes the stream resulting from invoking `ps_ax` repeatedly at consecutive logical time steps. It is analogous to the Unix filter `yes` that repeatedly returns its argument over time. Even though all GLU streams are conceptually infinite, a special end-of-data value `eod` is used to denote the end of finite streams. GLU provides special operations that manipulate streams such as `fby` for combining two streams, `next` to drop the first value of a stream, `wvr` to select values from a stream, and `upon` to fill out a stream. These operations work not only on temporal streams but also on multidimensional streams. The stream processing capabilities of GLU can be used to express cyclic compositions as well as data-parallel compositions.

One of the high-level aspects of GLU is that GLU compositions express parallelism implicitly. This unburdens the programmer from having to explicitly manage parallelism on a per-application basis. The parallelism in GLU compositions is exploited using a novel model of execution — namely, demand-driven execution (also known as *eduction*) [1, 2]. The demand-driven model is general in that all forms of parallelism can be discovered and exploited. Unlike data-driven execution, demand-driven execution only causes useful filter functions to be invoked, thereby avoiding superfluous computations.

2.2 Translation of GLU Programs

The architecture of the GLU programming system is shown in Figure 1.

The GLU program is translated to an abstract machine, one that embodies the eduction model of computation. The abstract machine program representation of the GLU program consists of two parts: generator and worker. The generator consists of an application-indecedent abstract machine code interpreter, the abstract machine code for the GLU composition being translated, sequential functions to be locally invoked, and stubs for remotely invoking sequential functions. The worker

consists of sequential functions that will be invoked by the generator with suitable wrappers. The interaction between generator and workers is assumed to be using a high-level remote procedure call facility.

The generator-worker representation of the GLU program can be compiled to a variety of parallel architectures — including workstation network, shared-memory multiprocessor, and massively parallel processors — provided the remote procedure call facility is available on the individual architectures. This compilation is conventional since the generator and the worker are C programs.

Executing a compiled GLU program requires execution of the generator and invocation of one or more workers. The generator continually generates executable functions (functions with available arguments) by interpreting the GLU composition using the abstract machine code interpreter. These functions are executed at the remote workers and the results are passed back to the generator. Note that it is possible to specify at compile-time or run-time that a function should be executed in the generator itself — functions so specified are executed in the generator while all other functions are executed at the worker. The principal source of parallelism is the simultaneous invocation of functions in multiple workers.

2.3 Related Approaches

GLU offers a high-level alternative to explicit, low-level approaches to parallel programming. Notable among the low-level approaches are PVM [8], Linda [3], and varieties of remote procedure call and message-passing packages [9]. The essential difference between GLU and these approaches is that GLU unburdens the programmer from the responsibility of discovering and managing parallelism. Thus, issues such as partitioning, mapping, scheduling, and load balancing are largely transparent to the programmer. Without this transparency, parallel programming becomes quite difficult as the programmer is forced to deal with issues at different levels of abstraction at the same time. The main criticism of high-level approaches such as GLU is that they are not efficient when compared to their low-level counterparts. Our experience with prototypical parallel applications suggests otherwise. Besides, we believe that the success of parallel programming will be initially determined more by how easy it is to build applications and less by how efficient they are.

3 Construction of a GLU Application: An Example

Consider the problem of multiplying two matrices, say A and B, and obtaining a product matrix C. The standard solution is to obtain each element (i, j) of the product matrix C by multiplying the i^{th} row of A with the j^{th} column of B and summing the inner product. A procedural program (in the language C) for multiplying matrices is as follows:

```
#include <stdio.h>

struct matrix {
        int nrows, ncols;
        float **v;
        };
typedef struct matrix *MATRIX;


MATRIX block( MATRIX M, int frow, int lrow, int fcol, int lcol )
{
  MATRIX m;
  int i, j;

  m = (MATRIX) malloc( sizeof( struct matrix ) );
  m->nrows = lrow - frow + 1;  m->ncols = lcol - fcol + 1;
  m->v = (float **) malloc( m->nrows * sizeof( float * ) );
  for( i=0; i<m->nrows; i++ )
    m->v[i] = (float *) malloc( m->ncols * sizeof( float ) );
```

```
  for( i=0; i<m->nrows; i++ )
    for( j=0; j<m->ncols; j++ )
      m->v[i][j] = M->v[frow+i][fcol+j];

  return( m );
}

MATRIX mult( MATRIX a, MATRIX b )
{
  MATRIX m;
  int i, j, k;
  float s;

  m = (MATRIX) malloc( sizeof( struct matrix ) );
  m->nrows = a->nrows;  m->ncols = b->ncols;
  m->v = (float **) malloc( m->nrows * sizeof( float * ) );
  for( i=0; i<m->nrows; i++ )
    m->v[i] = (float *) malloc( m->ncols * sizeof( float ) );

  for( i=0; i<m->nrows; i++ )
    for( j=0; j<m->ncols; j++ )
      {
        s = 0;
        for( k=0; k<a->ncols; k++ )
          s = s + a->v[i][k]*b->v[k][j];
        m->v[i][j] = s;
      }

  return( m );

}

MATRIX matmult( MATRIX A, MATRIX B )
{
 int i, j, nr, nc;
 MATRIX C, r;

 nr = A->nrows;
 nc = B->ncols;
 C = (MATRIX) malloc( sizeof( struct matrix ) );
 C->nrows = nr; C->ncols = nc;
 C->v = (float **) malloc( nr*sizeof( float * ) );
 for( i=0; i<nr; i++ )
   C->v[i] = (float *) malloc( nc*sizeof( float ) );

 for( i=0; i<nr; i++ )
   for( j=0; j<nc; j++ )
     {
        r = mult( block(A,i,i,0,nc-1), block(B,0,nr-1,j,j) );
        C->v[i][j] = r->v[0][0];
      }

 return( C );
}

main()
{
  MATRIX A, B, C;
```

```
  MATRIX inA(), inB();
  void out();

  A = inA();
  B = inB();
  C = matmult( A, B );
  out( C );
}
```

The only user-defined datatype in the above program is `MATRIX`, which is a pointer to the structure `matrix` that consists of the number of rows, number of columns, and a two-dimensional array of `floats`.

In the above program, functions `inA` and `inB` are used to input matrices and `out` is used to output the product matrix; function `block` is used to extract a submatrix from a matrix; function `mult` is used to multiply compatible submatrices; and function `matmult` uses these functions to multiply two matrices.

We can develop a variant of function `matmult` as follows:

```
MATRIX gather( MATRIX tl, tr, bl, br )
{
  MATRIX m;
  int i, j, hnr, hnc;

  m = (MATRIX ) malloc( sizeof( struct matrix  ) );
  m->nrows = tl->nrows*2; m->ncols = tl->ncols*2;
  m->v = (float **) malloc(m->nrows*sizeof(float *));
  for(i=0; i<m->nrows; i++)
    m->v[i] = (float *) malloc(m->ncols * sizeof(float));

  hnr = m->nrows / 2;
  hnc = m->ncols / 2;

  for(i=0; i<m->nrows; i++)
    for(j=0; j<m->ncols; j++)
      {
        if( i < hnr )
          { if( j < hnc )
              m->v[i][j] = tl->v[i][j];
            else
              m->v[i][j] = tr->v[i][j-hnc];
          }
        else
          { if( j < hnc )
              m->v[i][j] = bl->v[i-hnr][j];
            else
              m->v[i][j] = br->v[i-hnr][j-hnc];
          }
      }

  return( m );
}

MATRIX lefthalf( MATRIX A )
{
  return( block( A, 0, A->nrows-1, 0, A->ncols/2 - 1 ) );
}
```

```
MATRIX righthalf( MATRIX A )
{
  return( block( A, 0, A->nrows-1, A->ncols/2, A->ncols-1 ) );
}

MATRIX tophalf( MATRIX A )
{
  return( block( A, 0, A->nrows/2 - 1, 0, A->ncols-1 ) );
}

MATRIX bottomhalf( MATRIX A )
{
  return( block( A, A->nrows/2, A->nrows-1, 0, A->ncols-1 ) );
}

MATRIX matmult( MATRIX A, MATRIX B )
{
 MATRIX C, TL, TR, BL, BR;

 TL = mult( lefthalf( A ), tophalf( B ) );
 TR = mult( righthalf( A ), tophalf( B ) );
 BL = mult( lefthalf( A ), bottomhalf( B ) );
 BR = mult( righthalf( A ), bottomhalf( B ) );

 C = gather( TL, TR, BL, BR );

 return( C );
}
```

In this program, multiplying a matrix consists of simply gathering the submatrices obtained by multiplying the appropriate half-matrices. The gathering function is performed using function `gather`, while selecting the appropriate half-matrices is performed using functions `lefthalf`, `righthalf`, `tophalf`, and `bottomhalf`, each of which is defined in terms of `block`.

The following routine constructs this program:

```
struct matrix {
        int nrows, ncols;
        float **m;
        }
typedef matrix *MATRIX;

int out( MATRIX );
MATRIX inA( );
MATRIX inB( );
MATRIX gather( MATRIX, MATRIX, MATRIX, MATRIX );
MATRIX lefthalf( MATRIX );
MATRIX righthalf( MATRIX );
MATRIX tophalf( MATRIX );
MATRIX bottomhalf( MATRIX );
MATRIX mult( MATRIX, MATRIX );

out( C ) fby eod where
  A = inA( );
  B = inB( );
  C = matmult( A, B );

  matmult( A, B ) = P where
     P = gather( TL, TR, BL, BR );
```

```
    TL = mult( lefthalf( A ), tophalf( B ) );
    TR = mult( righthalf( A ), tophalf( B ) );
    BL = mult( lefthalf( A ), bottomhalf( B ) );
    BR = mult( righthalf( A ), bottomhalf( B ) );
  end;
end
```

The GLU program has two parts – the first part consists of user-defined datatypes and user-defined function-prototype declarations, and the second part consists of the composition of the program using built-in and user-defined functions. The syntax of the first part is identical to ANSI C, and hence needs no further explanation. The composition is simply an expression, in this case `out( C ) fby eod` where the enclosing *where clause* defines the variable `C`.

The expression `out(C) fby eod` defines a temporal sequence whose first element is the result of invoking `out(C)`, which has the side-effect of displaying matrix `C`, and whose subsequent elements are all `eod`, which is a special value denoting end of data. With most GLU interpreters, the operational effect of displaying an `eod` is termination of program execution.

The definition of `C` is given in the enclosing *where clause* as `matmult( A, B )`, where variables `A` and `B` themselves are defined using user-defined function `in`. Unlike `out` and `in` (which are C functions), `matmult` is a user-defined GLU function with two parameters. Function `matmult` is defined to be the variable `P` where `P` itself is the result of applying `gather` to `TL`, `TR`, `BL`, and `BR`. Variable `TL` is the result of multiplying (using `mult`) the left half of matrix `A` (obtained using `lefthalf( A )`) and the top half of matrix `B` (obtained using `tophalf( B )`). Variables `TR`, `BL`, and `BR` are similarly defined.

It is worth noting how this composition is evaluated, given that it is declarative and not procedural. The first thing that happens is that expression `out(C) fby eod` is evaluated. It causes `out(C)` to be evaluated first and `eod` to be evaluated next. Since `out` is a procedural user-defined function, it is invoked only when variable `C` is defined and no sooner. To evaluate `C`, the right-hand side of its definition, `matmult( A, B )`, is evaluated. Since `matmult` is a composition function and not a procedural function, it is evaluated even before its arguments `A` and `B` are available. In fact, all composition functions are evaluated using "call-by-need" semantics as opposed to "call-by-value" semantics used for C functions.

The main difference between a GLU program and an equivalent procedural program is that the GLU program is inherently parallel unless otherwise constrained, whereas the procedural program is inherently sequential unless parallelism is explicitly identified. In the example above, with the GLU program, the variables `TL`, `TR`, `BL`, and `BR` can be computed simultaneously, resulting in fourfold parallelism, whereas these variables have to be evaluated sequentially in the procedural program.

Since GLU programs are declarative, they possess the important property of *referential transparency*, which means that two occurrences of an expression refer to the same value. Thus, the order of equations in GLU programs is simply a matter of style and does not affect their meanings. One can substitute the right-hand side of an equation for each occurrence of the left-hand side and vice versa, just as in mathematics.

GLU programs separate composition from computation. This is important in various aspects of the program development process, such as debugging and modification, and also encourages reuse of existing computational code across applications.

The GLU program described above simply multiplies two matrices and then terminates. What if we want to have the program multiply two streams of matrices pairwise? It turns out that doing so in GLU is not that much more work than multiplying just one pair. The GLU composition for this procedure is

```
struct matrix {
        int nrows, ncols;
```

```
        float **m;
        }
typedef matrix *MATRIX;

int out( MATRIX );
MATRIX inA( int );
MATRIX inB( int );
MATRIX gather( MATRIX, MATRIX, MATRIX, MATRIX );
MATRIX lefthalf( MATRIX );
MATRIX righthalf( MATRIX );
MATRIX tophalf( MATRIX );
MATRIX bottomhalf( MATRIX );
MATRIX mult( MATRIX, MATRIX );

out( C ) where
  A = inA( #.time );
  B = inB( #.time );
  C = matmult( A, B );

  matmult( A, B ) = P where
     P = gather( TL, TR, BL, BR );
     TL = mult( lefthalf( A ), tophalf( B ) );
     TR = mult( righthalf( A ), tophalf( B ) );
     BL = mult( lefthalf( A ), bottomhalf( B ) );
     BR = mult( righthalf( A ), bottomhalf( B ) );
  end;
end
```

This program differs from the previous one in only two respects: functions `inA` and `inB` now have an argument, `#.time`, and the program expression is simply `out(C)` instead of `out(C) fby eod`.

What this program reveals is the underlying context in which GLU programs are evaluated — namely, `time`. That is, each expression (including variables) in the above GLU program actually denotes a temporal stream of values. The value of an expression depends not only on the constituent subexpressions and their binding function but also on the implicit time context. Thus, one can think of the equation `C = matmult( A, B )` as meaning

$\texttt{<C}_0\texttt{,...,C}_t\texttt{,...> = <matmult(A,B)}_0\texttt{,...,matmult(A,B)}_t\texttt{,...>}$
where $\texttt{matmult(A,B)}_t$ `= matmult(` $\texttt{A}_t$`,` $\texttt{B}_t$ `).`

Consider the evaluation of $\texttt{out(C)}_t$: it causes $\texttt{C}_t$ to be evaluated, which causes $\texttt{matmult(A,B)}_t$ to be evaluated, which, in turn, causes $\texttt{gather(TL}_t\texttt{,TR}_t\texttt{,BL}_t\texttt{,BR}_t\texttt{)}$ to be evaluated. Evaluations of $\texttt{TL}_t$,$\texttt{TR}_t$,$\texttt{BL}_t$, and $\texttt{BR}_t$ eventually cause both $\texttt{A}_t$ and $\texttt{B}_t$ to be evaluated once. Evaluation of $\texttt{A}_t$ invokes `inA` with `#.time` being the current time context — namely, t. (And similarly for $\texttt{B}_t$.) Once $\texttt{A}_t$ and $\texttt{B}_t$ are available, $\texttt{TL}_t$,$\texttt{TR}_t$,$\texttt{BL}_t$, and $\texttt{BR}_t$ can be generated, resulting in $\texttt{C}_t$ to be generated and function `out` to display the matrix.

Since evaluation of $\texttt{C}_t$ is independent of $\texttt{C}_{t+1}$, it is entirely possible for both these computations and any others to proceed simultaneously without interference.

Suppose we wanted to multiply only the first 10 matrix pairs and then terminate – this requires changing only the program expression from `out(C)` to `if #.time < 10 then out(C) else eod fi`.

One of the limitations of `matmult` is that it results in only fourfold parallelism since function `mult` is sequential. It is possible to have `matmult` express more parallelism by making `matmult` recursive:

```
  matmult( A, B ) =
    if smallenough(A,B)
```

```
  then mult(A,B)
  else P fi where
   P = gather( TL, TR, BL, BR );
   TL = matmult( lefthalf( A ), tophalf( B ) );
   TR = matmult( righthalf( A ), tophalf( B ) );
   BL = matmult( lefthalf( A ), bottomhalf( B ) );
   BR = matmult( righthalf( A ), bottomhalf( B ) );
end;
```

In this program, we have introduced a function `smallenough` that helps control the granularity of parallelism by deciding when multiplying two matrices cannot be further subdivided. Composition function `matmult` itself is recursive – instead of having `TL`, `TR`, `BL`, and `BR` defined in terms of `mult`, these variables are defined using `matmult`. The parallelism expressed by `matmult` is no longer fourfold — it is actually $O(4^L)$, where L is the number of recursive steps.

A drawback of the recursive `matmult` is the cost associated with repeated division of matrices using `lefthalf`, `righthalf`, `tophalf`, and `bottomhalf`. This not only takes time but also causes unnecessary creation and destruction of intermediate matrices. We devise a nonrecursive version of `matmult` that expresses as much parallelism, without the cost of recursion. To do so, we use the GLU notion of user-defined multidimensionality:

```
matmult( A, B ) = p asa.t ( (nrows(A) == nrows(p)) && (ncols(B) == ncols(p)) )
  where
    dimension i,j,t;
    a = rowstrip( A, #.i );
    b = colstrip( B, #.j );
    p = mult( a, b ) fby.t
        ( ( gather( p, next.j p, next.i p, next.i next.j p )
        @.j (2*#.j) ) @.i (2*#.i) );
  end;
```

Composition function `matmult` is defined using a multidimensional computation in dimensions `i`, `j`, and `t`, which are declared using the `dimension` declaration within the *where clause* associated with `matmult`. This multidimensional computation occurs at each `time`-context in the enclosing `time` dimension. It is defined to be the value of variable `p` at context `i=0,j=0,t=T` where `T` is the `t`-context at which the number of rows of the product matrix `p` is the same as the number of rows of matrix `A` and the number of columns is the same as the number of columns of matrix `B`.

Consider the definition of variable `p`, which says that at any context where `t`-context is 0, it is the value of `mult(a,b)` at the same context. Variable `a`, which varies only in dimension `i`, is defined as function `rowstrip(A, #.i)`, where `#.i` refers to the `i`-context from the context of evaluation. (`rowstrip` basically extracts the $\texttt{i}^{th}$ strip of rows from matrix `a`.) Variable `b`, which varies only in dimension `j`, is defined as function `colstrip(B, #.j)` where `#.j` refers to the `j`-context from the context of evaluation. (It extracts the $\texttt{j}^{th}$ strip of columns from matrix `b`.)

Variable `p` at context `(i=I,j=J,t=T)` where `T>0` is the result of applying function `gather` to values of variable `p` at contexts `(i=2*I,j=2*J,t=T-1)`, `(i=2*I,j=2*J+1,t=T-1)`, `(i=2*I+1,j=2*J,t=T-1)`, and `(i=2*I+1,j=2*J+1,t=T-1)`. This can be derived as follows:

$$
\begin{aligned}
&\texttt{P(i=I,j=J,t=T)} = \\
&\quad \texttt{(gather(p,next.j p,next.i p,next.i next.j)@.j(2*\#.j))@.i(2*\#.i)}_{\texttt{(i=I,j=J,t=T-1)}} \\
&= \texttt{gather(p,next.j p,next.i p,next.i next.j)}_{\texttt{(i=2*I,j=2*J,t=T-1)}} \\
&= \texttt{gather(P}_{\texttt{(i=2*I,j=2*J,t=T-1)}}\texttt{,P}_{\texttt{(i=2*I,j=2*J+1,t=T-1)}}\texttt{,} \\
&\qquad \texttt{P}_{\texttt{(i=2*I+1,j=2*J,t=T-1)}}\texttt{,P}_{\texttt{(i=2*I+1,j=2*J+1,t=T-1)}}\texttt{)}
\end{aligned}
$$

The best way to visualize the evaluation of `matmult` is to think of each of its variables as representing a series of planes (`i`,`j`) ordered by time `t`. Variable `a` then is a series of identical planes

(since the definition of `a` does not depend on time `t`) where each plane consists of identical columns of row strips where each row strip is extracted from matrix `a`. Similarly, variable `b` is a series of identical planes where each plane consists of identical rows of column strips where each column strip is extracted from matrix `b`. (The row strip size and column strip size are predefined.) It is worth noting that any reasonable implementation of GLU would store values of `a` for each `i`-context and values of `b` for each `j`-context, as they are constant in the remaining two dimensions.

Variable `p` consists of a series of planes, where the initial plane consists of matrix products such that the $(i,j)^{th}$ matrix is obtained by multiplying the corresponding (actually i^{th}) row strip of `a` and corresponding (actually j^{th}) column strip of `b`. Each successive plane consists of matrices that are four times as large where the $(i,j)^{th}$ matrix is obtained by coalescing the four matrices using function `gather` in the previous plane at positions $(2i,2j)$, $(2i,2j+1)$, $(2i+1,2j)$, and $(2i+1,2j+1)$.

Assuming that the size of each element matrix in the initial plane is s and the size of the product matrix is S, the $(0,0)^{th}$ element of the $\log_2(S/s)$ plane is the desired product matrix. One way of determining without keeping track of size is to check whether the $(0,0)^{th}$ element matrix of a plane has the same number of rows as matrix `a` and the same number of columns as matrix `b`, as in `p asa.t ( (nrows(A) == nrows(p)) && (ncols(B) == ncols(p)) )`.

Similar to the recursive version of `matmult`, the parallelism in simultaneous execution of `mult` is 4^L where L is the number of planes. (L is $\log_2(S/s)$.)

It turns out that this kind of computational structure is common to algorithms that solve quite different problems. Therefore, it would be desirable to specify the structure in such a way that it can be not only general but also succinct. This is the idea behind what we call dimensionally abstract composition functions — functions that abstract not only data but also dimensions. We illustrate one such function, `planar_tree`, in `matmult`:

```
planar_tree.x,y( f, a, size ) = b asa.t s >= size
  where
    dimension t;
    b = a fby.t
        ( ( f( b, next.y b, next.x b, next.y next.x b )
        @.x (2 * #.x)) @.y (2 * #.y));
    s = 1 fby.t 4*s;
  end;
matmult( A, B ) = planar_tree.i,j( gather, mult(a, b), n )
  where
    dimension i,j;
    a = rowstrip( A, #.i );
    b = colstrip( B, #.j );
    n = ( nrows(A) / ROWSTRIPSIZE ) * ( nrows(B) / COLSTRIPSIZE );
  end;
```

In `matmult`, `planar_tree` is invoked with dimensions `x` and `y` corresponding to `i` and `j`, `f` corresponding to data function `gather`, `a` corresponding to `mult(a,b)`, and `m` corresponding to `n`.

4 Application Experience

In this section, we describe our experience in converting two existing sequential applications to GLU and their resulting performance improvement when executed on a network of workstations. It is worth noting that other sequential applications have similarly been converted to GLU including a 400,000 line message handling system [6] and a large structural engineering code [7, 2].

4.1 Parallel Make

Make, a widely used Unix utility, is a natural candidate for parallelization. In fact, there are several commercial and public domain offerings of parallel **make**. The idea in **make** is to use a set

of acyclic dependencies between program objects (usually files) to determine whether a target object is current; if the target object is not current, to use the associated actions to build a current target object.

This obviously is a recursive definition as each such object, in turn, is a target with its own set of dependencies. Associated with each dependency is a set of actions that builds a target if it is not current.

We describe the parallelization of **make** (specifically **GnuMake**) by converting to GLU to illustrate the rapidity and effectiveness with which we were able to do so. (Interestingly, the **make** algorithm and the GLU model of computation have much in common — both are demand-driven and both operate on data-dependency networks.)

We started with **GnuMake**, which is a sequentially written application from Free Software Foundation. It consists of over 15,000 lines of C code in 35 files. The principal source of parallelism is in being able to build multiple independent targets simultaneously. This parallelism is latent in the makefile itself. In fact, **GnuMake** provides the capability to exploit this parallelism on a shared-memory multiprocessor by starting multiple processes to build independent targets.

We are interested in creating a parallel make that exhibits the same behavior as **GnuMake**. And we want to do so without writing large amounts of new code. Our approach is simple to describe. We modified **GnuMake** to generate dependency information of the targets that would have been built without actually building them. This information is piped to a GLU program that uses it to construct the dependency tree. Starting with the root of the tree, it applies the associated build actions after each of its immediate ancestors have been successfully built and this proceeds recursively. The GLU program composition is shown below.

```
typedef int CODE;
typedef int STATUS;
typedef struct cmds {
    int         count;
    string      line[count];
    } *CMDS;
typedef struct object {
    string      name;
    int         no_dependents;
    CMDS        commands;
    void        *treeptr;
    } *OBJECT;
local int       input_from_make( int );
local OBJECT    getobject( int );
local int       getancestor( int, int );
      STATUS    apply( OBJECT, CODE );

M( input_from_make( 0 ) ) fby eod
where
  dimension target;
  M( f ) = if iseod f then -1 else make fi;
  make = if errcode then errcode else apply(object, errcode) fi;
  object = getobject( #.target );
  errcode = done where
    dimension sibling;
    done = if ( iseod ancestors ) then 0
           else (make @.target ancestors) | next.sibling done fi;
    ancestors = if a<0 then eod else a fi where
            a = getancestor(#.target, #.sibling);
        end;
    end;
end
```

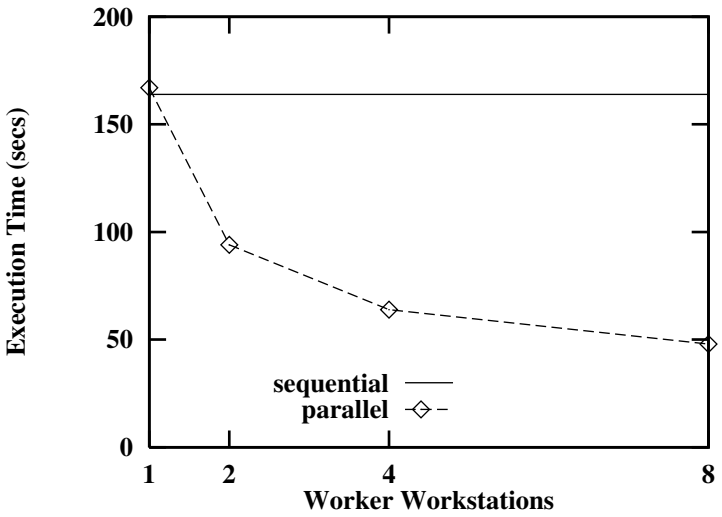

Figure 2: Performance of Parallel GLU Make

It uses four C functions `input_from_make`, `getobject`, `getancestor` and `apply`.. Function `input_from_make` constructs a dependency tree using information provided by a benign invocation of **GnuMake**. The dependency tree is kept in a global data structure (actually, a hash table) accessible to `local` functions i.e., ones that execute in the generator itself. Function `getobject` returns an `OBJECT` data structure pointed to by its argument from the hashtable. Function `getancestor` returns the index of the appropriate ancestor of the current target (given by `#.target`). Function `apply` builds the current target using actions given in its first argument. It is the only function executed in the workers; all other functions being `local` are executed in the generator itself.

The program corresponds to GLU function `M(input_from_make(0))` which has the effect of building the dependency tree in a hashtable and initiating the make process starting with the root target (at `#.target=0`). For `apply(object,errcode)` to be invoked at context `#.target=0`, `getobject` and `errcode` at the same context must be evaluated. Function `getobject` will return immediately but evaluation of `errcode` requires further evaluation. In particular, it requires all the immediate ancestors of target at `#.target=0` to be evaluated in the same way. And this proceeds recursively to the leaves of the dependency tree. At a leaf target, since there are no ancestors, its `errcode` can be immediately computed causing its `apply` to be invoked. The results of such `apply` invocations will flow back to other suspended `apply` invocations, and eventually to the one associated with the root target (`#.target=0`).

The conversion only required adding 15 lines of C code and changing 2 lines of C code in GnuMake. The GLU make program itself is less than 40 lines of composition and 250 lines of new C code for the sequential functions declared in the composition.

Figure 2 shows the performance of parallel GLU make applied to a typical makefile (for **GnuEmacs** generation) on a ethernet-based network of lightly loaded, similarly configured Sun SS2 workstations with a common file system. The speedup using parallel make is best seen for a small number of workstations. It drops off as the number of workstations increase. This is partly because the granularity of parallelism (ratio of remote function time to communication overhead) is small and variable (as builds of different targets take different times). The other reason is that the root of the dependency tree has a monolithic build that takes sizable time (25 seconds in the

experiment). This adversely affects the extent to which performance can scale with additional workstations.

4.2 Parallel Raytracing

Raytracing is a widely used graphics application that is inherently highly parallel. The basic idea in raytracing is to render a two-dimensional view of an observer of a 3D scene consisting of objects with different reflective and refractive properties and multiple sources of light. The two-dimensional 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 independent of the others.

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 two-dimensional 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 in the workers; all other functions being `local` are executed in the generator itself.

The GLU composition 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 two-dimensional pyramid computation (specified by `p` shown in Figure 3).

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 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.

Figure 4 shows the performance of parallel raytracing constructing a series of 400x400 pixel image on a network of Sun SS2 workstations. It scales much better than parallel make and exhibits

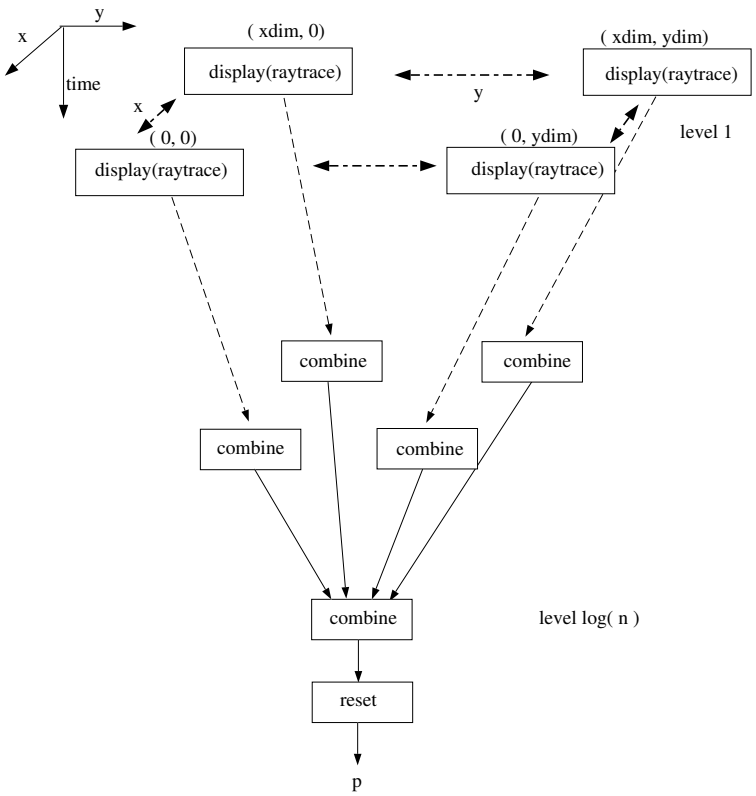

Figure 3: Two-Dimensional Raytracing Computational Pyramid

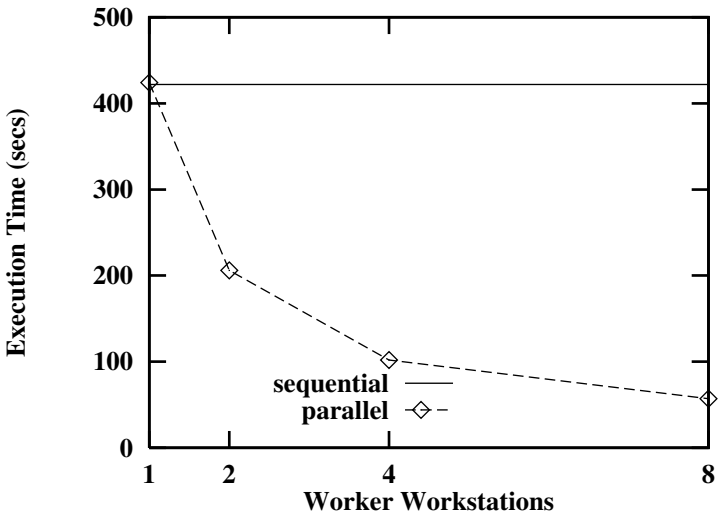

Figure 4: Performance of Parallel Raytracing

near-linear speedup for the number of workstations we considered. This application performs better than parallel make because the granularity of parallelism is much coarser and quite uniform. In addition, parallel make of the makefile we considered has a sizable sequential part as we noted earlier.

5 Conclusions

Our experience with a very high-level composition language called GLU demonstrates its rôle in constructing parallel applications using existing sequentially written codes. The expressiveness of the language is illustrated by the different ways in which a prototypical sequentially written, inherently parallel application (matrix multiplication) can be parallelized.

Two widely used, sequentially written applications — namely, **make** and raytracing — have been converted to parallel GLU equivalents with only a nominal amount of modifications and additions. Application speed has impressively improved by running parallel versions of the two applications on a network of workstations.

A very high-level language like GLU offers parallel programmers the ability to solve problems without getting mired in the often intricate details of how to manage parallelism; as is often the case when using low-level parallel languages. In addition, it enables existing sequential applications to be rapidly converted to parellel GLU applications by allowing most of the existing sequential code to be reused.

Acknowledgements

This work was supported in part by the National Science Foundation (Grant Number CCR-9203732) and in part by SRI's IR&D investment program.

References

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

[2] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional Declarative Programming*. Oxford University Press, 1994.

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

[4] 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.

[5] R. Jagannathan and A.A. Faustini. GLU: A hybrid language for parallel applications programming. Technical Report SRI-CSL-92-13, Computer Science Laboratory, SRI International, Menlo Park, California 94025, 1992.

[6] Y. Koui, S. Seno, T. Yamauchi, M. Ishizaka, and K Kotaka. Implementation and evaluation of MHS parallel processing. *IEICE Transactions on Communications*, E77B(11), November 1994. Tokyo, Japan.

[7] E. Moreno. GLU BRIDGE—A structural engineering application in the GLU programming language. Master's thesis, Arizona State University, Tempe, Arizona 85287, U.S.A., 1993.

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

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

Author Information

R. Jagannathan is a Senior Computer Scientist at SRI International in the Computer Science Laboratory. His interests include declarative languages, parallel and distributed computing, and adaptive systems. He received a B.Sc. from the University of Calgary in 1979 and a M.Math. and a Ph.D. from the University of Waterloo in 1981 and 1988 respectively.

Chris Dodd is a Senior Software Engineer at SRI International in the Computer Science Laboratory. His interests include compilation techniques, parallel and distributed operating systems, and languages. He received a B.S. from the California Institute of Technology in 1988 and a M.S. from Stanford University in 1990.