LaRCS: A Language for Describing Parallel Computations for the Purpose of Mapping [*]

Virginia M. Lo[†], Sanjay Rajopadhye[‡], Moataz A. Mohamed[§]
Samik Gupta, Bill Nitzberg, Jan Arne Telle, Xiaoxiong Zhong
Dept. of Computer and Information Science
University of Oregon
Eugene, Oregon 97403-1202
email: lo@cs.uoregon.edu

Abstract

LaRCS is a graph description language which enables the programmer of parallel algorithms to specify information about the static and temporal communication behavior of parallel algorithms. The information contained in a LaRCS program is used for the mapping problem: to assign tasks in the parallel computation to processors, and to route inter-task messages along the links of the interconnection network. Many practical algorithms exhibit regular communication patterns and LaRCS provides a mechanism for describing this regularity in a compact, parameterized manner. Static communication topology is expressed in LaRCS code through node labels and simple communication functions on the nodes. Temporal communication behavior is represented by a notation we have developed called *phase expressions*.

This paper introduces the LaRCS language and compares LaRCS with several existing graph description languages. We introduce a new graph theoretic model of parallel computation based on LaRCS called the **T**emporal **C**ommunication **G**raph; and we show how information provided by LaRCS plays an important role in mapping.

---

[*] This research was sponsored by Oregon Advanced Computing Institute (OACIS), a consortium of academic, industrial, and government agencies in the state of Oregon.

[†] Partially supported by NSF grant CCR-8808532

[‡] Partially supported by NSF grant MIP-8802454

[§] Moataz Mohamed is currently a member of Computer Systems Lab, Stanford University

Keywords: configuration language, graph description language, regular parallel computation, mapping, parallel programming environments.

Contents

1 Introduction

The *mapping problem* in message-passing parallel processors involves the assignment of tasks in a parallel computation to processors and the routing of inter-task messages along the links of the interconnection network. Most commercial parallel processing systems today rely on manual task assignment by the programmer and message routing that does not utilize information about the communication patterns of the computation. The goal of our research is *automatic* and *guided* mapping of parallel computations to parallel architectures in order to achieve portability and maximal performance from parallel software.

We have developed a unified set of software tools for automatic mapping of parallel computations to parallel architectures. Our system is called OREGAMI[1]. Two key features of our system are (a) the ability to take advantage of the regularity present in both the computation structure and the interconnection network and (b) the desire to balance the user's knowledge and intuition with the computational power of efficient combinatorial algorithms.

In this paper, we focus on one component of the OREGAMI system called **LaRCS** (**La**nguage for **R**egular **C**ommunication **S**tructures). LaRCS is a description language which enables the programmer of parallel algorithms to specify information about the static communication topology and temporal communication behavior of the computation to be mapped. The design of LaRCS was driven by the observation that many parallel computations exhibit regularity in both static communication topology and temporal communication patterns. LaRCS can be used to capture this regularity for use by OREGAMI's mapping algorithms.

Fig. 1 illustrates the relationship of LaRCS to the components of the OREGAMI system, MAPPER and METRICS. MAPPER, our mapping software, is a library of algorithms which map the computation to the architecture. The specific algorithm that is invoked depends on the information provided by the LaRCS description. METRICS is an interactive graphics tool which displays the mapping along with a range of performance metrics reflecting load balancing, communication contention, and communication overhead.

LaRCS is designed for use in conjunction with parallel programming languages in

---

[1] For University of **OREG**on's contribution to the elegant symmetric structures (contractions) produced by ori**GAMI** paper folding.

Figure 1: Overview of OREGAMI

which the programmer designs his or her computation as a set of communicating parallel processes. Candidates include OCCAM, C*, Data Parallel C, and C and Fortran with communication extensions. The underlying architecture is assumed to consist of heterogeneous processors connected by some regular network topology. Note, however, that OREGAMI is a front-end mapping tool and generates symbolic mapping directives only. Development of back-end software to transform OREGAMI mapping directives to architecture dependent code is beyond the current scope of our work.

Our software tools prototype is implemented in MacScheme and Think-C on a MacII using color displays for visualization of the mapping. An updated version of OREGAMI written in C for the Open Windows interface on Sun workstations should be available for distribution Fall 1991. Additional information about MAPPER and METRICS can be found in [LRG$^+$90b].

In this paper, we discuss the features of LaRCS and illustrate its ability to describe a wide range of parallel computations. In Sec. 2 we describe the key language features found in LaRCS: `nodetypes`, `comtypes`, `comphases`, and `phase expressions`. In Sec. 3, we describe the graph theoretic model of parallel computation underlying LaRCS and show how it integrates two well-known existing models of parallel computation. In Sec. 4, we compare LaRCS to existing graph description languages and configuration languages, and we discuss the strengths and limitations of LaRCS. In Sec. 5, we show how information provided by LaRCS plays an important role in mapping. In Sec. 6, we conclude by discussing areas of on-going and future work.

2 LaRCS Overview

The LaRCS graph description language enables the programmer to describe the static communication topology and temporal communication behavior of the parallel computation in a compact and intuitive way. In particular, LaRCS allows the user to conveniently express any *regularity* exhibited by the parallel computation. We say a **family** of graphs is *regular* if it can be parameterized and can be described by a finite set of rules the size of which is independent of the parameter values. The design of LaRCS was guided by the following principles:

- **Abstraction.** LaRCS should provide a clean abstraction to represent only those characteristics of the parallel computation of interest to the mapping algorithms, namely the regular communication structure, both static and temporal.
- **Compactness.** The size of the LaRCS description of a family of graphs should be constant, independent of the size of the task graph. The different members of the family should be obtained simply by appropriate instantiations of the parameters.
- **Portability.** LaRCS should be capable of being used with a wide variety of parallel programming languages and with a wide range of interconnection network topologies.
- **Simplicity.** LaRCS should be easy to understand and use, and the language constructs should be natural and intuitive.

2.1 LaRCS Language Features

The LaRCS abstraction of a parallel program is a collection of processes which are involved in two activities: computation and communication (message-passing). In graph theoretic terms, processes are represented nodes and communication is represented by directed edges between nodes. A LaRCS program consists of the following major components: (a) the LaRCS `nodetype` declaration which describes the processes, (b) the LaRCS `comtype` and `comphase` declarations which are used as templates to describe the communication structure of the parallel program, and (c) the LaRCS `phase expression` which describes the entire parallel program in terms of its temporal computation and communication behavior. A `comtype` declaration describes a single communication edge; a `comphase` describes a set of synchronous communication edges. A `phase expression` instantiates all the edges of a parallel computation using the `comtype` and `comphase` declarations, and describes the message-passing behavior of the computation over time. We note that there is a distinction between "temporal communication behavior" (as used here) and dynamically evolving task graphs. In our model the entire graph is known statically. We describe temporal behavior by identifying collections of edges that are "active" simultaneously, and by the pattern of this activity. The conventional static task graph used in the research on mapping algorithms to processors [Sto77] can be defined as the union of all instantiated edges in a LaRCS program. The model of parallel computation underlying LaRCS is described more fully in Sec. 3.

We now describe the components of a LaRCS program, using a parallel algorithm for the *n-body problem* as an example. This algorithm was developed for the Caltech Cosmic Cube [Sei85]. Fig. 2 gives a description of the the *n-body problem* and Fig. 3 gives the LaRCS code.

1. **Name of algorithm and parameters.** The parameters specify the size of this instance of the parallel algorithm. The parameters for the *n-body algorithm* are `n`, the number of bodies, and `s`, the number of iterations[2].

2. **Attributes.** The programmer may specify global characteristics of the task graph, such as `nodesymmetric` or `planar`.

3. **Nodetype declaration.** A `nodetype` is defined by giving it a name, specifying the number of nodes, and specifying the node labeling. Node labels can be multidimensional and parameterized. For the *n-body algorithm* there is one `nodetype` declaration of type `body`. The nodes are labeled from `0` to `n-1`. If there is only one `nodetype`, the explicit declaration may be omitted.

4. **Comtype declaration.** A `comtype` specifies a single potential edge and can be parameterized. In both `comtype` and `comphase` declarations the symbol `=>` denotes unidirectional message passing and `<=>` denotes bidirectional message passing. In Fig. 3, there are two `comtype` declarations: `ring_edge` and `chordal_edge`. The `volume` field of the `comtype` declaration is an expression which specifies the message volume (typically in bytes) of a single message transfer.

5. **Comphase declaration.** A `comphase` identifies a potential set of edges involved in synchronous message passing, usually by specifying a set of values for the parameter(s) of one or more `comtypes`. The `comphase` declaration may itself be parameterized and these parameters are later instantiated within the phase expression. In the *n-body algorithm* there are two `comphase` declarations: `ring` and `chordal`.

---

[2] Note that since LaRCS is intended for static mapping, we require an estimate of the run-time parameter, `s`.

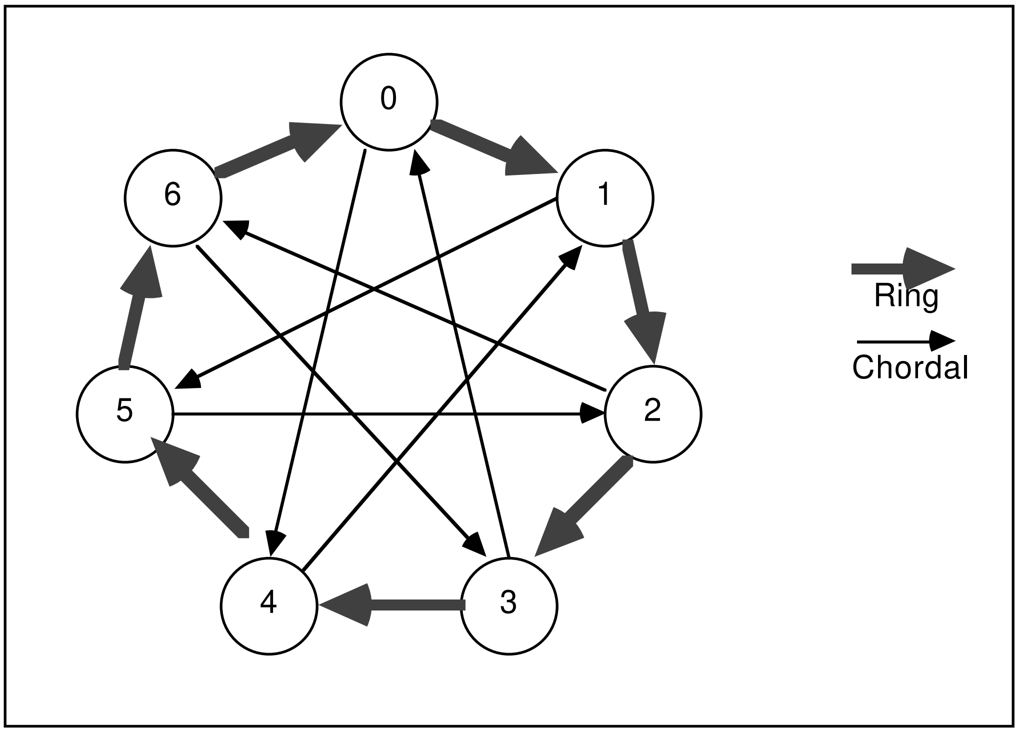


Figure 2: Task graph for the *n-body algorithm*

The *n-body problem* requires determining the equilibrium of n bodies in space (where n is odd) under the action of a (gravitational, electrostatic, etc.) field. This is done iteratively by computing the net force exerted on each body by the others (given their "current" position), updating its location based on this force, and repeating this until the forces are as close to zero as desired. The parallel algorithm presented by Seitz uses Newton's third law of motion to avoid duplication of effort in the force computation. It consists of n identical tasks, each one responsible for one body. The tasks are arranged in a ring and pass information about their accumulated forces to its neighbor around the ring. After $(n-1)/2$ steps, each task will have received information from half of its predecessors around the ring. Each task then acquires information about the remaining bodies by receiving a message from its chordal neighbor halfway around the ring. This is repeated to the desired degree of accuracy.

```
# LaRCS code for the n-body problem

nbody(n,s)
attributes nodesymmetric;
nodetype body
    labels 0..(n-1);
comtype ring_edge(i) body(i) => body((i+1) mod n);
comtype chordal_edge(i) body(i) => body((i+(n+1)/2) mod n);
comphase ring
     forall i in 0..(n-1) {ring_edge(i);}
comphase chordal
     forall i in 0..(n-1) {chordal_edge(i);}
phase_expr
     {{ring |> compute}**(n-1)/2 |> chordal |> compute}**s;
```

Figure 3: LaRCS code for the n-body problem

The `comtype` declaration is optional. For our example, the `comphase` declarations could have been written as shown below.

```
comphase ring forall i in 0..n-1
            {body(i) => body((i+1) mod n);}
comphase chordal forall i in 0..n-1
            {body(i) => body((i+ (n+1)/2) mod n);}
```

6. **Phase expression.** The phase expression describes the temporal behavior of the computation in terms of its communication phases. For the *n-body problem*, the phase expression is given by:

```
{{ring |> compute}**(n+1)/2 |> chordal |> compute}**s;
```

Phase expressions are defined recursively below where r and s are phase expressions.

- `compute` is a keyword phase expression denoting a computational phase of activity.

- a single `comphase` is a phase expression.
- sequence: r `|>` s is a phase expressions which denotes sequential execution of the phases.
- sequential repetition: r `**` *expr* is a phase expression denoting repeated execution of r a number of times specified by arithmetic expression *expr*.
- sequential loop: `for var =` *range* $\{ r \}$ is a phase expression denoting repeated execution of r a number of times specified by *range*, where `var` is a formal paprameter in r.
- parallelism: r `||` s is a phase expression denoting parallel execution of phases r and s.
- parameterized parallelism: `forall` *var* `in` *range* $\{ r \}$ is a phase expression denoting parallel execution of phases in r, where *var* appears as a parameter in r.

Only expressions derived by a finite application of these rules are phase expressions.

2.2 Examples of LaRCS Programs

LaRCS has been used to describe a wide variety of parallel algorithms including several algorithms for matrix multiplication, fast Fourier transform, topological sort, divide and conquer using binary tree, divide and conquer using binomial tree, simulated annealing, Jacobi iterative method for solving Laplace equations on a rectangle, successive-over-relaxation iterative method, perfect broadcast distributed voting, numeric integration, distributed dot product, five point difference operation, Gaussian elimination with partial pivoting, matrix row rotation, and the simplex algorithm. We have found that LaRCS is particularly well-suited to data-parallel algorithms in which all processes perform identical operations on distributed data. Our study of the above algorithms has both influenced the design of LaRCS as well as served as an evaluation of the expressive power of the language.

In this section we present two additional examples of LaRCS code: a master-slave ring-structured application and the binomial tree divide-and-conquer parallel programming paradigm. Additional examples are given in Section 4.

Master-slave ring-structured application

A set of master processes is shadowed by a set of slave processes, both configured in the same ring structure. Periodically, each of the master processes sends control information to the corresponding slave process; the master and slave processes then continue to compute independently, with each set of processes sending messages around its own ring. See Figure 4 for a graphical representation of this computation and the corresponding LaRCS code.

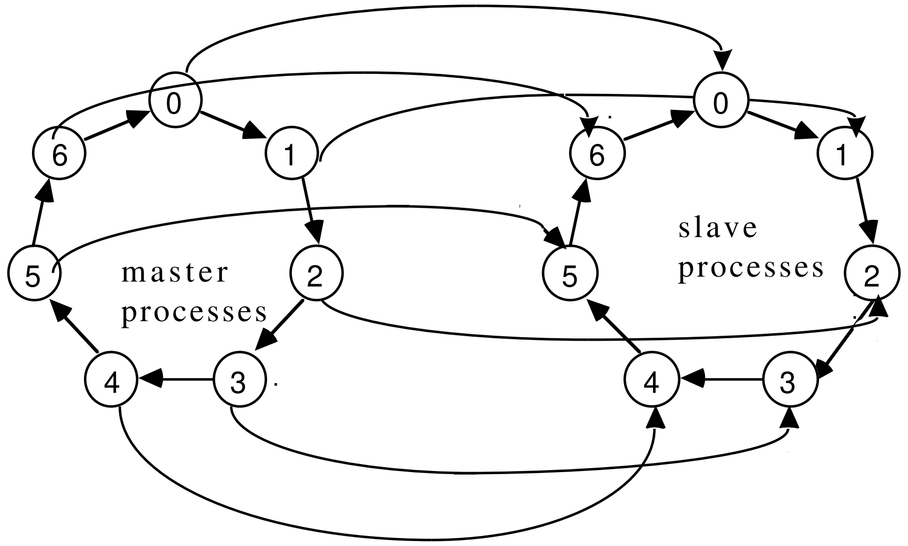
The LaRCS code for this problem illustrates the use of multiple `nodetype` declarations and generic `comtypes`. Generic `comtypes` allow the user to define a communication structure and then to associate that structure with more than one set of processes (`nodetype`). In this example, we declare the generic `comtype ring` and then declare two instances: `master_ring` and `slave_ring`.

Divide-and-conquer binomial tree in LaRCS

The binomial tree has been shown to be an excellent task graph for efficient computation of parallel divide and conquer applications [Vui87] [LRG$^+$90a]. The binomial tree is defined recursively as follows:

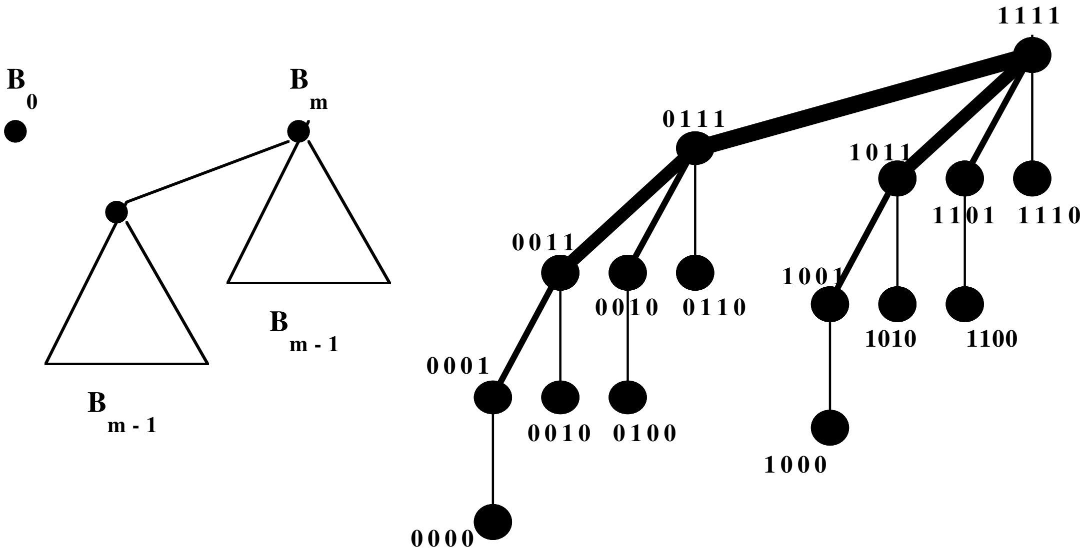
- The binomial tree B_0 consists of a single node.
- The binomial tree B_n is formed by connecting the root nodes of two identical binomial trees B_{n-1}. One of the root nodes is selected as the root for B_n.

A 16-node binomial tree and the LaRCS code for the general divide-and-conquer binomial tree are shown in Fig. 5. Here, the size of the binomial tree is specified by the number of `levels` in the tree. The binomial tree is labeled in postorder. The declaration for `comtype child(i,p)` indicates that in phase `p`, each node with binary label `i` communicates with the node whose label is computed by changing bit `p` of the binary label `i` from `1` to `0`, where the msb is bit 0 and the first phase is 0. This level of detail is needed to capture the temporal behavior of the algorithm. The parameterized *volume* expression associated with each `comtype` indicates that half of the problem is sent to each child process.



```
master_slave(n, s, k)                    # n is the size of the ring
                                         # s and k are iterations
nodetype master(n)
        labels 0..n-1;
nodetype slave(n)
        labels 0..n-1;
comtype ring(type,i)                    # A generic comtype, used to
         type(i)  => type ((i+1) mod n);   # define both master_ring and
                                        # slave_ring
comphase master_ring
        forall i in 0..n-1 {ring(master,i) ; }
comphase slave_ring
        forall i in 0..n-1 {ring(slave,i) ; }
comphase checkpoint
        forall i in 0..n-1 {master(i)  => slave(i) ; }
phase_expr
        master_ring  **s  |>
        {checkpoint  |>  {master_rinmg**s  ||  slave_ring**s }}**k  |>
        slave_ring;
```

Figure 4: Master/slave ring application



```
div_conq_binomial_tree(levels)
nodelabels  0..2**levels-1;

comtype child(i,p) i => i - 2**(levels - (p+1));
                    volume = 1/ 2**(p+1);       # if algorithm is Sort
comtype parent(i,p)  i - 2**(levels - (p+1)) => i;
                    volume = 1/ 2**(p+1);

comphase  split(phase)  forall  node  in  0..2**phase-1
                    {child(((node+1)*2**(levels-phase) -1), phase) ; }
comphase  merge(phase)  forall  node  in  0..2**phase-1
                    {parent(((node+1)*2**(levels-phase) -1), phase) ; }

phase_expr                                      # split, sort, merge
       for k = 0 to levels-1   {complute |> split(k) } |>
       compute |>
       for k = levels-1 to  0 {merge(k) |> compute};
```

Figure 5: Divide and Conquer binomial tree

3 The TCG Model of Parallel Computation

The graph theoretic model of a parallel computation underlying LaRCS is called the temporal communication graph (TCG). Both the TCG and its forerunner, the static task graph model of Stone [Sto77], were designed for systems in which the programmer designs his or her program as a set of *static* and *persistent* parallel processes which communicate through explicit message-passing. The processes are *static* in that they are created at compile-time; they are *persistent* in that they exist throughout the lifetime of the computation. The TCG models additional information about the temporal behavior of each of the processes, yielding a DAG representation of the parallel computation which retains the identity of each of the (persistent) processes comprising the computation. Finally, the TCG provides the ability to describe *regularity* in both the communication topology and the temporal communication behavior. In terms of LaRCS, the `nodetype`, `comtype`, and `comphase` declarations describe the static components of the TCG, while the `phase expression` defines the temporal behavior.

Thus, the TCG can be seen as a hybrid of the two predominant models of parallel computation: the static task graph model of Stone [Sto77], and the precedence-constrained (DAG) model [Pol88] used in multiprocessor scheduling and in the parallelization of sequential code. Task assignment and scheduling research utilizing these two models has more or less followed disjoint paths over the past two decades, in that techniques and algorithms developed for one model have not been applicable to the other. The TCG model is suitable for mapping and scheduling techniques from both domains. Thus algorithms for static task assignment such as [Bok87] [BS89] [Lo88], [SER90] and scheduling algorithms for precedence-constrained graphs such as [Pol88] [Bro86] [ERL90] can be applied to the TCG model. The TCG augments these two models with the ability to explicitly capture regularity, allowing the development of specialized mapping and scheduling algorithms to exploit this regularity.

Below is an intuitive definition of the TCG. A formal definition of the TCG and discussion of its utility in parallel programming environments is given in [Lo91].

The TCG abstraction defines a parallel computation as a collection of processes involved in two activities: computation and communication. A *compute phase* is a set of processes involved in *logically synchronous* computation. A *communication phase* is a set of pairs of processes (sender/receiver pairs) involved in *logically synchronous* communication. By *logically synchronous* we mean that the activities occur simultaneously from the viewpoint of the programmer, i.e. from the logical structural design of the algorithm. In reality, when the program executes, the timing of logically synchronous activities may not be synchronous with respect to real time, due to effects such as the hardware characteristics of the execution environment and the multiplexing of processes on the processors. In LaRCS *compute phases* are defined by the phase expression `compute` keyword;[3] *communication phases* are defined by the `comphase` declarations.

The TCG is constructed as follows. Because the construction involves three related graphs, we will use the terms *nodes* and *phase-nodes* in order to avoid confusion. (a) First consider the (static) task graph $T = < V, E >$ in which nodes correspond to the processes and directed edges to communication between processes. A compute phase corresponds to a subset of the processes that are active in that phase, i.e. to a subset of V. A communication phase corresponds to a subset of the communication edges, i.e., to a subset of E. Note that there may be several compute phases and several communication phases. (b) The phase-DAG describes the the logical temporal behavior of the computation in terms of its compute and communication phases. Thus, the phase-DAG consists of a set of phase-nodes in which each phase-node represents either a compute phase or a communication phase. The edges in the phase-DAG represent the precedence relationship among the compute and communication phases. (c) The TCG is an *elaboration* of the phase-DAG, in which each phase-node in the phase-DAG is replaced by the set of nodes or edges from the static task graph T involved in that phase.

This construction is illustrated for the *n-body problem*. The static task graph is shown in Figure 6(a). There are two compute phases, (compute1 and compute2). The two communication phases (ring and chordal) are represented by the solid and dotted edges, respectively. Figure 7(b) shows the phase-DAG which, for this example, is a linear chain of nodes. The number of phase-nodes in the chain is equal to the number of occurrences of the phases, as specified by the parameters $(n+1)/2$ and s. Figure 8(c) shows the TCG in which each phase-node of the phase-DAG is elaborated into the sets of process nodes and edges associated with each phase. Each (persistent) process is represented by the (vertical) chain of nodes appearing directly below the process numbers. The (vertical)

---

[3] The next version of LaRCS will include constructs for declaration of *compute phases* which will be the dual of the `comtype` and `comphase` declarations.

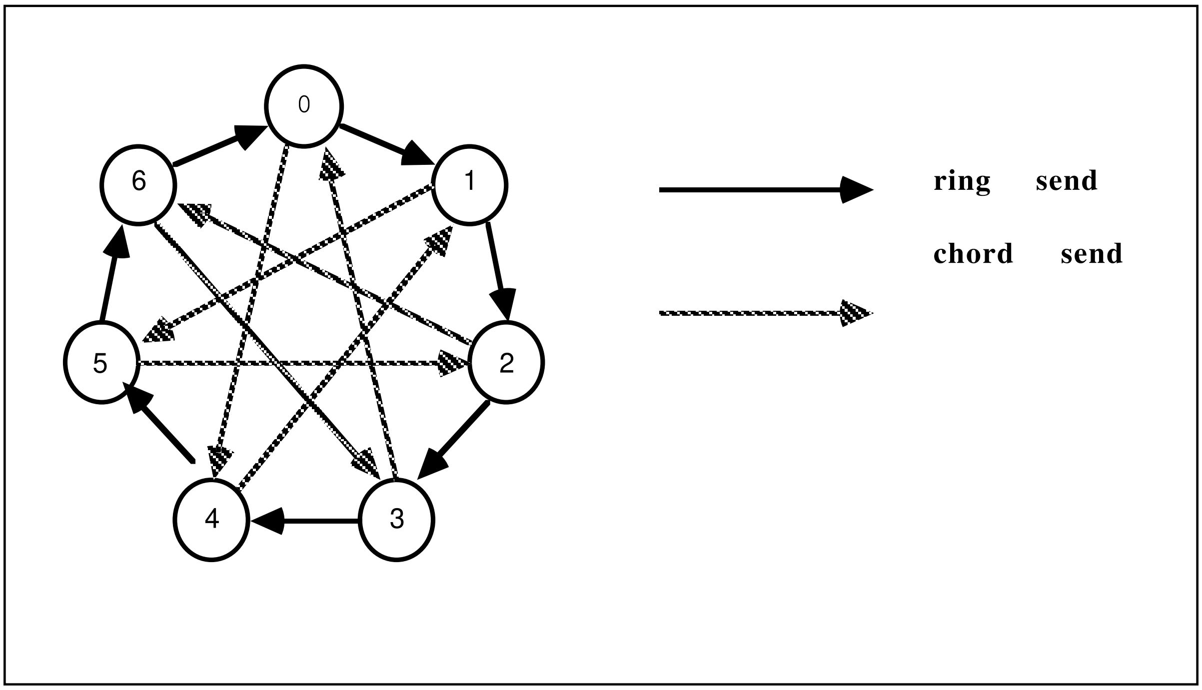


Figure 6: (a) Static task graph, compute phases, communication phases

edges between nodes in a chain specify the precedence relationship between the sequence of phases occurring within a single process. The other (non-vertical) edges correspond to the precedence relationship due to the message-passing that occurs in a communication phase. Thus, there are (vertical) directed edges from each instance of a `ring` node to the 'next' `compute1` node for that task, a (vertical) directed edge from the $(n+1)/2^{st}$ `compute1` node to the first `chordal` node, and (vertical) directed edges from each instance of a `chordal` node to the 'next' `compute2` node. The message passing event implied by an invocation of `ring` phase is represented by a directed edge from the the sender's `ring-send` node to the receiver's `ring-receive` node. The `chordal` phase is similarly represented by edges in the graph.

The TCG models both the static task graph view and the precedence-constrained DAG view of the parallel computation. The TCG can be seen as an unrolling of the static task graph over time. Conversely, the projection of the TCG along the time axis yields the static task graph model. Weights associated with the nodes and edges can be used to represent computation and communication costs, respectively. The means of expressing

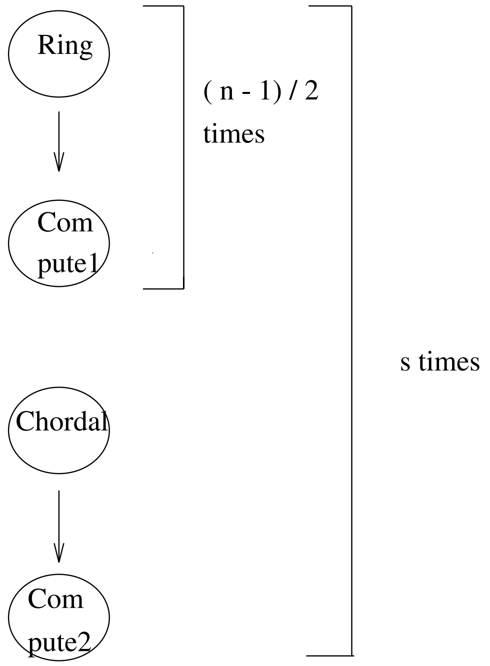


Figure 7: (b) Phase-DAG

edge and node weights using LaRCS is discussed in Sections 5.3 and 6.

We note that computations with irregular communication topology and asynchrony can also be modeled by the TCG and described by LaRCS. Any arbitrary set of communication edges can be decribed by defining a unique `comtype` for each edge. Asynchrony is modeled by defining a `comphase` to include the set of communication edges for which precedence is unknown, thereby increasing the granularity of the atomic unit used in the phase expression. Finally, we note that the TCG model does not have the capacity to model non-determinism, conditionals, or dynamically spawned processes although extensions for the latter case have already been developed and are under review internally.

4 A Comparison of LaRCS to Other Graph Description Languages

A number of researchers in the area of parallel and distributed computing have designed configuration languages or graph description languages. These languages provide a means for describing the processes and the communication topology of a parallel (distributed) computation. In the case of configuration languages the goal is typically the construction of a parallel computation from modular components, while in the case of graph description languages the goals include debugging and graphical display as well as the goal of LaRCS

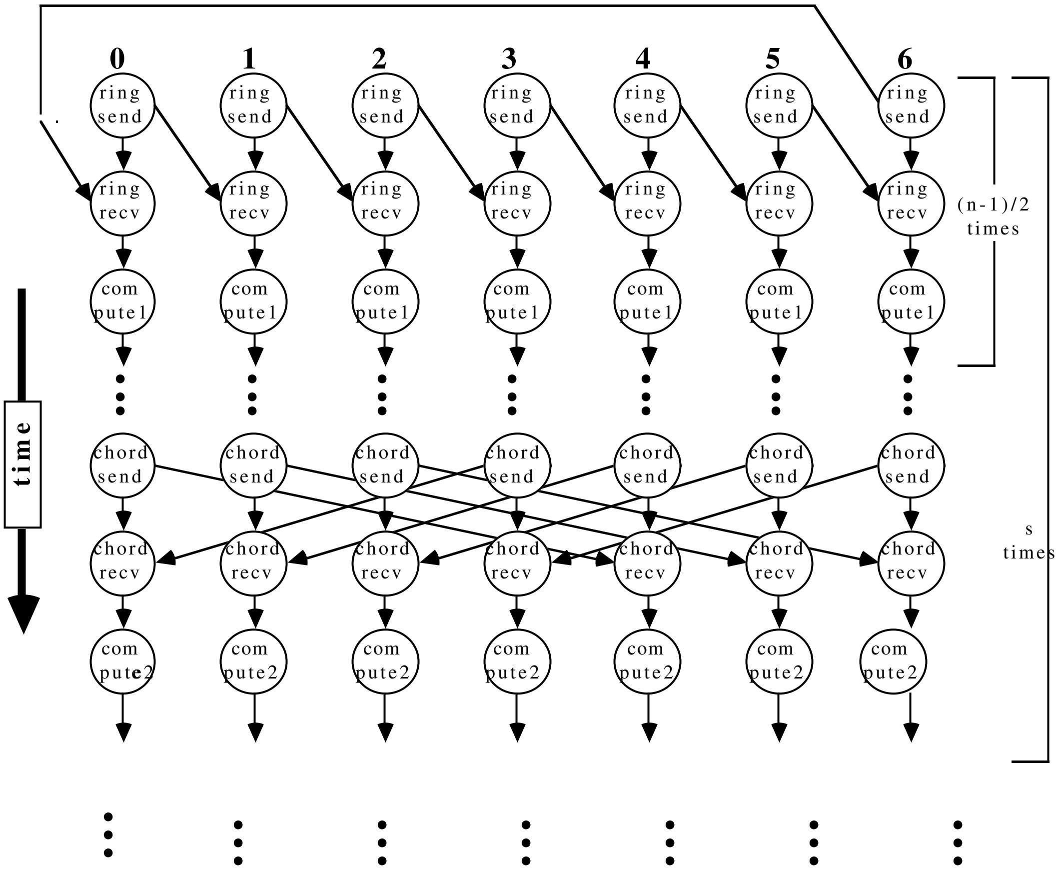


Figure 8: (c) TCG

```
                                  |  hypercube(d)
                                  |  nodelabels 0..(2**d-1);
Assert n is a power of 2          |  comphase links
link family k: [0..n-1],          |   forall j in 0..2**d-1
                d:[1..log2(n)]    |     {forall k in 0..d-1
A[k].xp to A[NegateBit (d,k)].ep; |       {j => j xor 2**k; }}
                                  | phase_expr links;
                                  |
```

Figure 9: A hypercube interconnection in Conic and LaRCS

which is to assist in the mapping of parallel algorithms to parallel architectures. In this section, we compare LaRCS to Conic, GDL, edge grammars, GARP and concurrent abstract grammars, ParaGraph and Attribute Rewrite Graph Grammars. Each of the comparisons highlights one or two specific features of LaRCS.

4.1 Conic

The Conic distributed programming environment [MKS89], developed at the Imperial College in London, provides tools for the hierarchical construction of distributed applications. The Conic programming language is a version of Pascal with extensions to support modularity and message-passing. The Conic configuration language is used at compile time to build the distributed application from smaller components by connecting the *exit ports* (`.xp`) of one module to the *entry ports* (`.ep`) of other modules. Conic configuration operations include `link` which specifies a uni-directional communication link, parameterized node labels, and a `broadcast` primitive. Fig. 9 shows the Conic and LaRCS descriptions of the hypercube interconnection network. This example was chosen to illustrate the ability of both languages to describe a static graph structure. Thus, this example does not represent the description of a parallel computation which is the more typical target of a Conic or LaRCS program.

- **Explicit vs. implicit communication ports.** Conic mandates that the programmer specify the communication exit and entry ports. Input and output ports are implicit in LaRCS.

- **Language dependence vs language independence.** Both Conic and GDL (described below) support a configuration language that is closely tied to an underlying programming language. LaRCS is designed to be used with a variety of parallel programming languages.
- **Implementation.** The Conic environment has been in use for a number of years for programming of distributed system applications.

4.2 Prep-P and GDL

Prep-P is an automatic mapping system developed by Berman [SB90] at UCSD which is quite similar to our OREGAMI system. Prep-P differs from OREGAMI in that the target architecture is a reconfigurable network, currently the CHiP network [Sny82]. The Prep-P system includes a parallel programming language 'XX' and a graph description language GDL. GDL is closely tied to 'XX' in the sense that a one-to-one correspondence exists between the procedures and ports declared in the 'XX' program and those used to describe the task graph in GDL. LaRCS and GDL share the same motivation: to provide a mechanism for describing the structure of parallel computations for the explicit purpose of mapping. In both, the graph is described by a program that is closely related to the computation being mapped.

In Fig. 10 we give the GDL description of a complete binary broadcast tree. The LaRCS program in Fig. 11 corresponds most closely to the GDL program in that they both describe a static tree configuration. In Fig. 12 we illustrate the ability of LaRCS to capture the actual temporal behavior of the level-by-level broadcast tree.

- **Temporal Communication Behavior.** The example in Fig. 12 illustrates the ability of LaRCS to describe the temporal communication behavior of a given parallel application. Both Conic and GDL are only capable of describing the static full binary tree. The ability to capture the temporal behavior of the communication edges is important for OREGAMI's routing algorithms. With the knowledge of which edges are simultaneously active, the routing algorithms can assign communication edges to links such that contention is minimized; scheduling and multiplexing can be coordinated.

```
irregular
nodemin=1
nodecount =15

procedure ROOT
 nodetype: {i=1}
 port LSON: {2*i}
 port RSON: {2*i +1}

procedure MID
 nodetype: {i>1 && i<(15+1)/2}
 port PARENT: {i/2}
 port LSON: {2*i}
 port RSON: {2*i +1}

procedure LEAF
 nodetype:  {i>=(15+1)/2}
 port PARENT: {i/2}
```

Figure 10: GDL full binary tree

```
static_binary_tree(levels)

nodelabels  1..2**levels - 1;

comtype left_child(i) i <=> 2*i;
comtype right_child(i) i <=> 2*i+1;

comphase full_tree
     forall k in 1..2**(levels - 1)-1
              {left_child(k); right_child(k);}

phase_expr        # all edges active simultaneously
     full_tree;
```

Figure 11: LaRCS full binary tree

```
# LaRCS temporal broadcast tree
level_by_level_binary_tree (levels)

nodelabels   1..2**levels - 1;

comtype  left_child(i) i => 2*i;
comtype  right_child(i) i => 2*i+1;

comphase  tree_band(k)          # all edges in one level
       forall node in 2**(k-1)..2**k-1
              {left_child(node); right_child(node);}

phase_expr                      # edges active level by level
for phase=1 to (levels-1) {tree_band(phase)};
```

Figure 12: LaRCS level-by-level broadcast tree

- **Node declarations.** In GDL, it is necessary to define a distinct nodetype for each process (`ROOT, MID, LEAF`). In LaRCS, it is not necessary to distinguish among the nodes, although its is possible to do so if desired. These differences arise from the fact that like Conic, GDL is tightly tied to the programming language 'XX' through the procedure declarations.

- **Edge declarations.** In GDL, edges are defined implicitly through the `PORT` declarations. Because the sender is always assumed to have the label `i`, the communication function and its inverse must both be specified. In LaRCS edges are defined explicitly, and the sender and receiver can both be described with arbitrary expressions.

- **Communication volume.** LaRCS allows the user to explicitly define the unit communication volume associated with a communication edge. In the Prep-P system, communication volume is estimated by the compiler from the program code.

4.3 Edge Grammars

Edge grammars were introduced by Berman [Ber83], [BS84], [BS87] as a means for describing graph families and as an abstraction for use in the mapping problem. Edge grammars define graph *families* by generating all the edges in the graph using conventional formal languages mechanisms. The edges of the graph are identified by the pairs of strings generated by the productions. The vertices of the graph are implicitly identified by those strings that appear in the pairs generated by some finite application of the productions. Given such a grammar, the sublanguage consisting of all k-length strings generated by the grammar defines a graph (the k-th member of a family of graphs).

Similar to conventional formal grammars, edge grammars have a hierarchy. The type 2 and type 3 edge grammars have been shown to be powerful enough to describe many commonly known interconnection structures.

Below are two examples of edge grammars: one for the complete binary tree and one for the hypercube. These grammars should be contrasted with the corresponding LaRCS code in Sections 4.2 and 4.1, respectively.

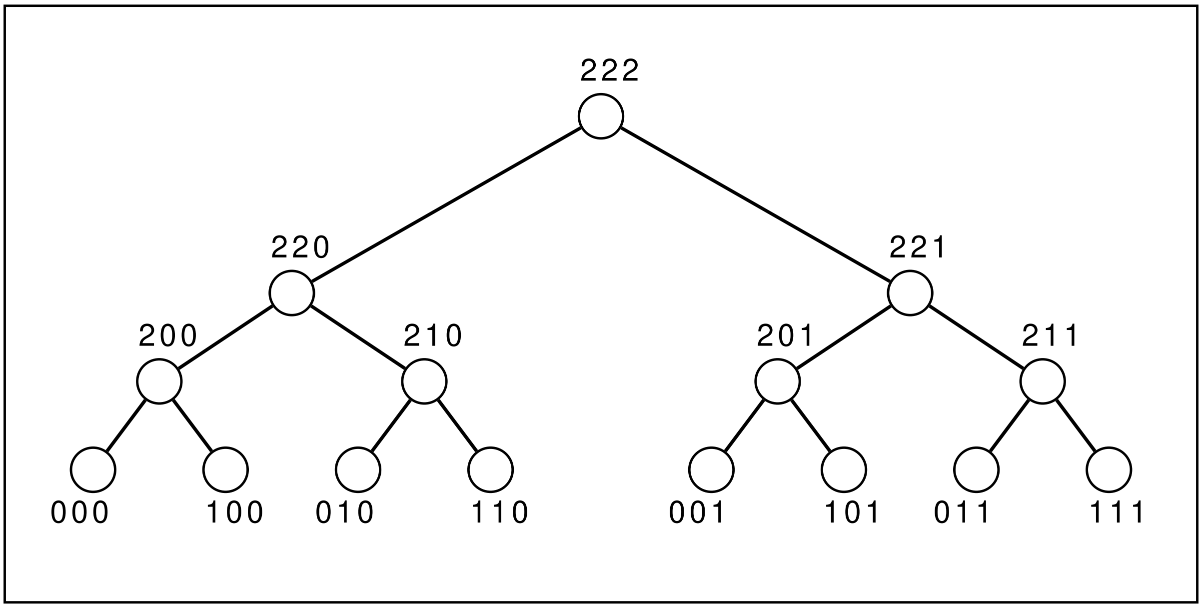


Figure 13: Binary tree generated by type 3 edge grammar

An edge grammar for complete binary trees

Berman and Snyder give the following type 3 edge grammar that generates the family of complete binary trees $\{B_n\}$. Let Γ be the edge grammar with nonterminals B_0, B_1, B, R, terminals (0,0), (1,1),(2,2),(2,0),(2,1), start symbol B, and the following productions:

$$\begin{array}{lll}
B \rightarrow R & B \rightarrow B_0 & B \rightarrow B_1 \\
B \rightarrow B_R & B_0 \rightarrow B(0,0) & B_1 \rightarrow B(1,1) \\
B_R \rightarrow R(2,0) & B_R \rightarrow R(2,1) & B \rightarrow (2,0) \\
B \rightarrow (2,1) & R \rightarrow (2,2) & R \rightarrow R(2,2)
\end{array}$$

The 15-node binary tree generated by this grammar is shown in Fig 13.

An edge grammar for Hypercubes

The nonterminals are S and M, where S is the start symbol. S generates pairs of strings that are identical in all except the last bit, and this is followed by strings generated by M, which generate the remaining matching bit patterns..

$S \rightarrow (0,0)S \quad M \rightarrow (0,0)M$
$S \rightarrow (1,1)S \quad M \rightarrow (1,1)M$
$S \rightarrow (0,1)M \quad M \rightarrow \epsilon$

Edge grammars are both elegant and useful for graph representation and for mapping. However, they have some disadvantages, some of which have been reported by the authors themselves [BS89]:

- **Inability to represent directed graphs.** As currently defined, the grammar does not provide any means for distinguishing between the source (sender) and the destination (receiver) of each edge.
- **No explicit specification of size of graphs.** The size of the generated graph is implicit since it is the number of times the productions are applied.
- **Difficult to use.** It is difficult for the average programmer to be able to come up with an edge grammar to describe the interconnection structure of his/her parallel programs.
- **Restricted contraction.** When used for contraction, edge grammars are restrictive because they contract a given graph to a smaller graph in the *same* family.

4.4 Concurrent Abstraction Graph Grammars

Graph grammars consist of rewrite rules where the left-hand side of a rewrite rule is a node in a graph, and the right-hand side is a graph along with an embedding rule of the form $L_v \rightarrow B_v, E_v$. When the rewrite rule is applied to a graph, say G, the node labelled L_v in the graph is replaced by the graph B_v and the remaining nodes of G are connected to B_v according to the embedding rule E_v. A graph can be generated using a graph grammar by starting with an initial node and applying the rewrite rules a finite number of times. A form of graph grammars called Concurrent Abstraction Grammars (CAGs) are used by the GARP system to specify parallel computations. GARP ($\underline{\text{G}}$raph $\underline{\text{A}}$bstractions for Concu$\underline{\text{r}}$rent $\underline{\text{P}}$rogramming) [KK88] is a model of parallel computation where tasks are spawned dynamically at runtime, and where the tasks that are spawned have a "regular" interconnection structure. The regularity of the subgraph of tasks that are spawned is described by a notation based on graph grammars. In GARP, every task has a rewrite rule associated with it which is invoked at **runtime** to replace that task (corresponding to the left-hand side of the rule) with a collection of tasks (corresponding to the right-hand side). The embedding rule specifies how the newly generated collection of tasks fits in with the rest of the task-graph. GARP provides the programmer with a graphical representation of the computation which is useful for visualization.

An example of a quicksort using CAGs in GARP [KK88] is shown in Fig. 14. The corresponding CAG specifies not just one graph, but a *set* of graphs. Which one of these graphs corresponds to the task graph of a particular computation becomes known only at run time when the rewrite rules are applied and the graph is generated. A LaRCS program also typically specifies a set of graphs, but the parameters passed to that program are known at compile time and so the task graph of the computation is known at compile time. One could write a LaRCS program that specifies the same set of graphs as specified by the CAG in Fig. 14 if the program were allowed to encode this run-time information as a parameter.

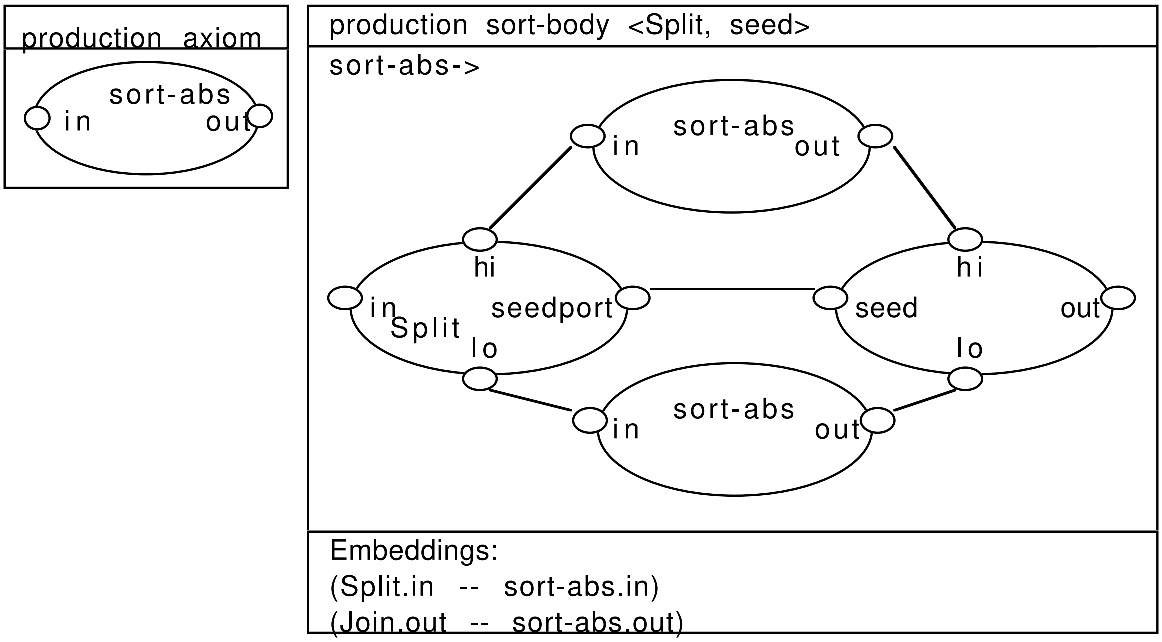
- **Expressive power.** As Kaplan and Kaiser themselves point out, there are many graphs for which it is difficult if not impossible to specify CAGs. We believe that this is so because of the limited options presented by the rewrite rules. Firstly, only nodes in a graph can be rewritten, not the edges. Secondly, even when rewriting nodes, there are only two kinds of embedding rules that can be used.

- **Dynamically evolving graphs.** CAGs (and hence GARP) allow the user to describe dynamically evolving task graphs. LaRCS is currently restricted to static graphs.

- **Inability to represent temporal communication behavior.** CAGs can only be used to describe the graph of the computation (including dynamically spawned subgraphs). However, It is not possible to specify which edges or nodes of that graph are being used at certain points in the computation.

4.5 Aggregate Rewriting Graph Grammars

An Aggregate Rewriting(AR) Grammar is a special type of graph grammars developed at the University of Massachusetts [BC87, BC89]. AR grammars are used to help the



This program takes a stream of numbers from standard input, sorts them using divide and conquer, and then passes the result to a standard output. The program executes recursively: When the sort-abs agent receives a message that is not the end-of-file object, it rewrites itself to a sort-body subgraph, passing the message just read as a seed value to the split vertex introduced in the rewrite rule. This split vertex passes all values received by it that are greater than the seed through the hi port, all other values through the lo port, and the seed itself through the seed port. The join agent waits for the messages on its lo, hi and seed ports and passes the concatenation of these three messages to its out port.

Figure 14: Sort example in GARP

programmer visually specify process interconnection structures in a parallel programming environment. This is achieved by a graphical editor called ParaGraph. The programmer can also annotate the interconnection structures (specified by an AR grammar) with code written in parallel programming languages that support explicit parallelism and interprocess communication.

An AR grammar consists of production rules which are applied to a graph called a host graph to derive a new graph. Unlike the rewrite rules in graph grammars used in the GARP system, a production rule in an AR grammar rewrites every occurence of a subgraph (called mother graph) with another graph (called daughter graph). The union of the occurences of the mother graph is called an Aggregate. Nodes of mother graphs as well as daughter graphs are labeled using a scheme defined by the user, and the edges of both kinds of the graphs are visually defined in ParaGraph. With the help of a partial function which maps the nodes in the daughter graph to those in the mother graph (called the inheritance function) and a partition of the domain of that function, the user can specify the connections between the daughter graph and the rest of the host graph. The language of an AR grammar represents a family of graphs. Recently, AR grammars have been used to describe some temporal computation patterns[BC89] such as pipeline patterns in a systolic matrix multiplication.

AR grammars have been used to describe several well-known interconnection structures including hypercubes, complete binary trees, back trees and butterfly structures. Fig. 15 shows an AR grammar which describes binary hypercube structures.

- **Expressive power.** A production rule in an AR grammar involves a node labeling scheme and the specification of the inheritance funcntion and the partition. These functions may be fairly complicated. Moreover, if the production rules are not applied in a correct order, an AR grammar may generate graphs which are not wanted.

- **Dynamically evolving graphs.** In [BC89], a facility called a *script* is used to denote a permissible sequence of transformations. The specification of scripts is an additional overhead for the user.

- **Hard to instantiate.** The parameters of graphs in an AR grammar are implicit. It is hard for the user to use the correct parameter(s) to instantiate the desired

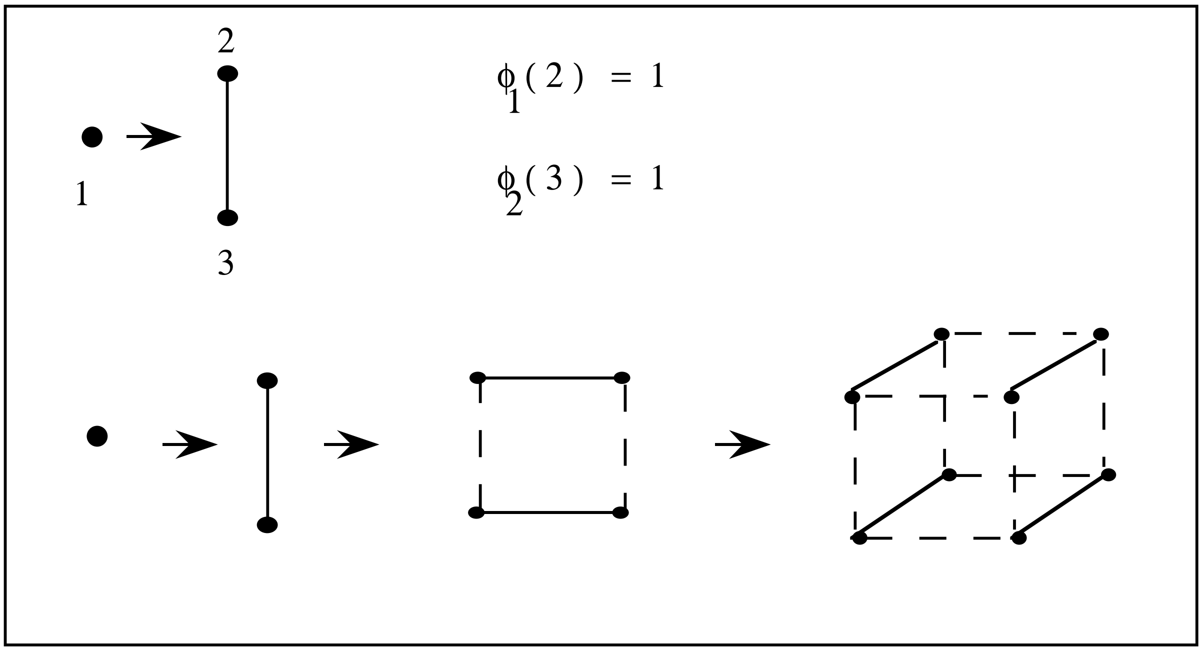


Figure 15: An AR grammar for binary hypercubes and some derivations

graph.

- **Representation of temporal communication behavior.** *Itineraries* in ParaGraph provide a means for describing the routing of specific data items (in messages) through the process graph.

4.6 Summary

Table 1 summarizes the key differences between LaRCS and the configuration languages we discussed above. Note that we are mainly interested in the mapping problem and hence our comparison is restricted to issues arising in this context.

Only OREGAMI and Prep-P are systems which directly address the mapping problem; thus, LaRCS and GDL are similar in many respects. LaRCS, GDL, and Conic enable the programmer to describe the computation graph using an algorithmic formalism which includes explicit communication functions. We believe this formalism is easier for the average programmer to use than the less intuitive grammar-based formalisms. LaRCS offers

	LaRCS	**GDL**	**Edge Grammars**	**Conic**	**CAG**	**AR**
System name.	OREGAMI	Prep-P	not implemented	Conic	GARP	ParaGraph
Underlying formalism.	algorithmic	algorithmic	edge grammars	algorithmic	graph grammars	graph grammars
Directly addresses mapping.	yes	yes	yes	no	no	no
Describes temporal behavior at compile time.	yes	no	no	no	no	limited
Describes dynamically evolving graphs.	no	no	no	no	yes	yes
Definition of ports.	implicit	explicit	implicit	explicit	explicit	explicit
Directed edges.	yes	yes	no	yes	no?	no
Weighted edges.	yes	limited	no	no	no	no

Table 1: A Comparison of Several Graph Description Languages

the unique ability to describe the temporal communication behavior of the computation through the phase expression. (The utility of this information for mapping is described Section 5.) Some of the graph description languages (GDL, Conic, CAG, and AR) are implemented within full blown programming languages or programming systems. LaRCS is not tied to a specific programming language because our current focus is the development of mapping algorithms applicable to a wide range of languages and architectures. Further strengths and limitations of LaRCS are discussed in Section 6.

5 The Role of LaRCS for Mapping

As discussed earlier, the mapping of a parallel computation to a parallel architecture involves the assignment of tasks to processors and the routing of messages through the underlying communication network. OREGAMI breaks the problem of mapping a task graph to a target architecture into three steps - contraction, embedding and routing. In this section we discuss ways LaRCS contributes directly to the operation of the OREGAMI mapping algorithms. We give examples to show how the comphase declaration plays a critical role in one of our contraction algorithms, how the comphase declarations are used by our routing algorithms to minimize contention; and how the phase expression can be used to derive accurate values of communication overhead and to guide estimation of the total completion time of the parallel computation. Finally, we discuss the general contribution of LaRCS to the mapping problem through its ability to represent families of regular computations in a space efficient way.

5.1 Contraction Based on LaRCS Comphases

We have developed an algorithm for contracting certain highly symmetrical task graphs in time quadratic in the number of tasks [LRG$^+$90b]. The algorithm is based in group theory and will yield a perfectly load balanced and symmetrical contraction whenever the task graph is a Cayley graph. All Cayley graphs are node-symmetric graphs, and we believe that in the message-passing paradigm it is common to partition a problem in such a way that the task graph becomes node-symmetric. It is a well-known fact that any quotient graph Q of a Cayley graph T is a perfectly load balanced and symmetrical contraction of T. No polynomial-time algorithm is known that recognizes Cayley graphs based on a general representation of the graph such as an adjacency matrix. However, given the LaRCS description of the task graph, in quadratic time we not only recognize whether it is a Cayley graph but also produce suitable contractions. The key step involves viewing the LaRCS comphases as possible generators of an underlying group. Assuming that only messages within one phase can contend for links, the resulting contraction is symmetrical because it evenly balances both the number of processes in a cluster and the number of messages from each phase on each link. Figure 16 illustrates our group theoretic contraction for the 8-node perfect broadcast parallel algorithm.

Many interesting interconnection networks, e.g. hypercubes, toruses, cube-connected-cycles, are themselves based on Cayley graphs [AK89]. We also plan to develop embedding, routing and multiplexing algorithms that exploit the common underpinnings of a Cayley task graph and these interconnection networks.

5.2 Routing Based on LaRCS Comphases

The LaRCS comphase declarations provide information that can be profitably used to achieve low contention routing in systems with user-level control over routing. Examples of such systems include Intel's iWarp Computer [ea90], the INMOS Transputer, Snyder's ChiP architecture, and some systems with store and forward message routing.

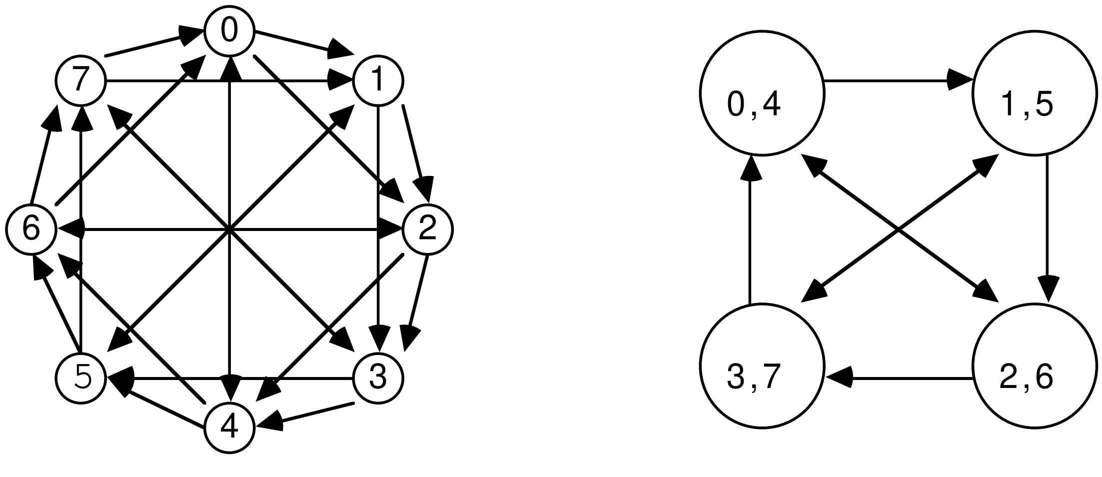
The comphase declaration identifies logically synchronous communication, i.e. message passing that can occur simultaneously at runtime. The OREGAMI routing algorithm [LRG$^+$90b] attempts to find a disjoint set of routes to service the set of messages belonging to a single `comphase`. This is accomplished by posing the routing problem as a bi-partite matching problem. In the bi-partite graph that is constructed by our routing algorithm, one partition consists of the communication edges in a single comphase and the other partition consists of the available (shortest) routes in the network that could potentially service these communication edges.

The information provided by the LaRCS comphase declaration enables the routing software to focus on only those messages that are capable of actual contention at runtime. There are several advantages to be gained by this approach:

- The likelihood of finding a routing with low contention is greater because fewer communication edges are considered.

```
nodelabels 0..7;
comphase comm1 task(i) => task((i+1) mod 8);
comphase comm2 task(i) => task((i+2) mod 8);
comphase comm3 task(i) => task((i+4) mod 8);
phase_expr
    comm1  |>  comm2  |>  comm3;
```

(a) Fragment of LaRCS code showing the communication types



(b) the task graph

(b) the contracted task graph

Figure 16: Group Theoretic Contraction

- The use of contention as a performance metric for mapping and embedding is more accurate, since we avoid measuring *false contention*, i.e., when two edges are mapped to the same link which are not active simultaneously at runtime.
- The use of the bi-partite matching algorithm avoids the contention problems of existing default routing algorithms.

Figure 17 illustrates the routing of a 6 process computation with a single comphase sendmsg which has been embedded in a 4 × 3 mesh. Figure 17(a) shows the OREGAMI routing which has no contention, while Figure 17(b) shows the routing that would result by using uninformed routing based on the iWarp machine's default streetsign mechanism.

5.3 The Role of the Phase Expression in Computing Communication Volume

Most existing mapping algorithms, including several in OREGAMI, utilize the *static task graph* and require an estimate of communication volume (edge weights.) Current techniques for computation of edge weights include profiling, user estimates, and compiler analysis of the program code. LaRCS provides a natural and convenient method for computing communication volume: the phase expression can be used to derive an arithmetic formula for calculation of communication volumes given the unit volume associated with a single message.

The LaRCS comtype-declaration's volume field can be initialized to the number of bytes sent in a single message of that comtype. For example, based on the code for the Caltech *n-body algorithm* given in [Sei85], the volume for the comtype ring is 108 bytes. This value would be specified by the user in the LaRCS code. The phase expression can be used to compute the total communication volume for each communication edge in the static task graph by multiplying the unit volume times the number of iterations derived from the phase expression. Thus, the total communication overhead for a ring edge of the *n-body algorithm* is equal to $108 \times n \times s$, where n and s are parameters instantiated at compile time.

At this stage in OREGAMI's development, the unit volume and all parameters must be expressed as integer constants; the phase expression then drives the calculation of the total communication volume for each comphase and for the complete static task graph.

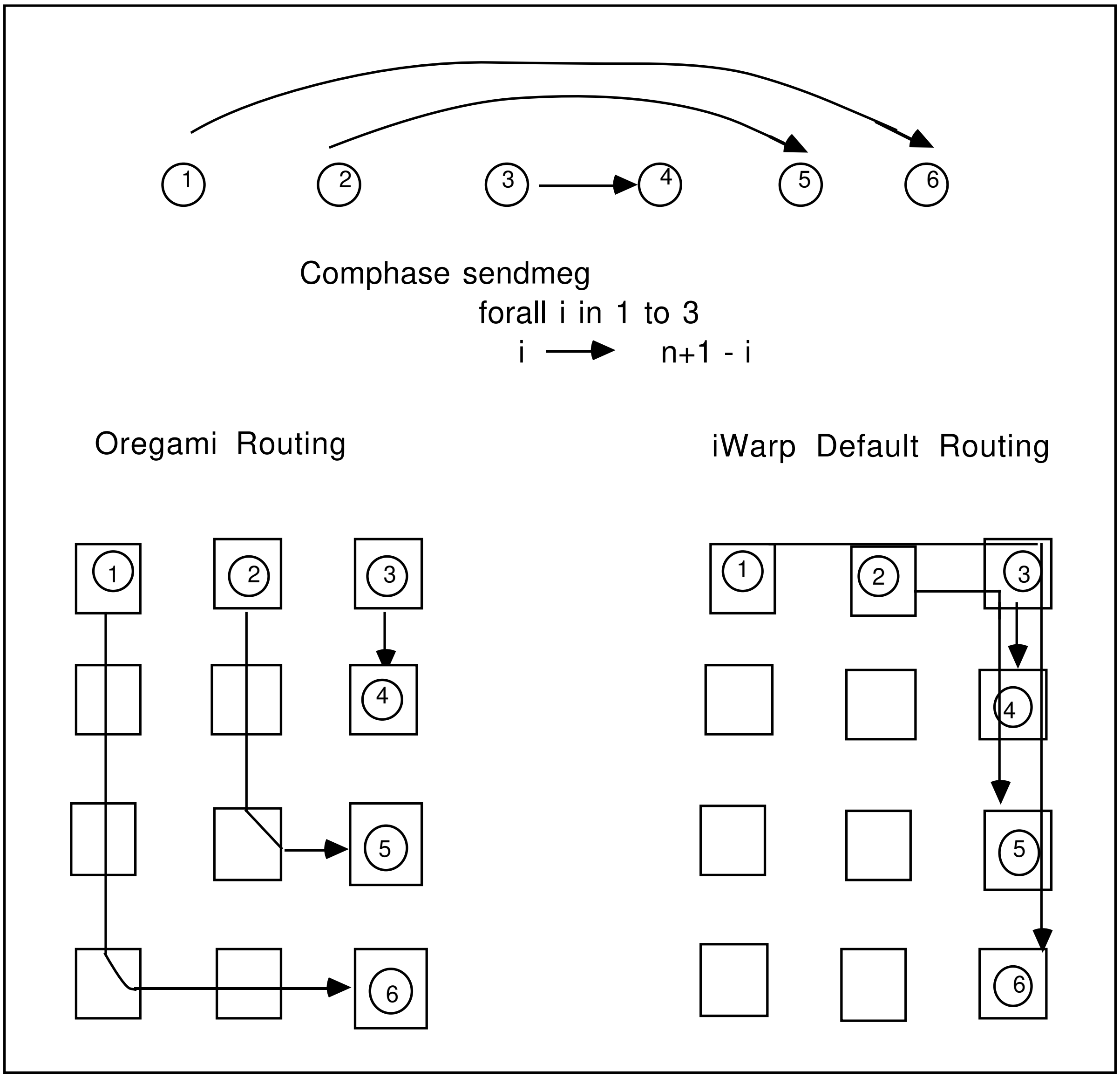


Figure 17: Routing with OREGAMI

In the future, we will be expanding our system to accept volume declarations in terms of imported variables from the host language. This extension relieves the programmer of the burden of counting up the number of bytes in a message involving complex data structures and/or multiple data structures. In addition, we will examine the feasibility of mapping which utilizes *volume expressions* with uninstantiated variables. To support mapping with *volume expressions*, OREGAMI would include algorithms for reducing these expressions to simpler form and mapping algorithms which utilize expressions rather than numeric edge weights.

5.4 The Role of the Phase Expression in Computing Completion Time

The phase expression is also useful in the evaluation of mapping algorithms based on the standard metric known as *completion time* (referred to as CT below). The METRICS component of OREGAMI estimates the completion time of a mapped computation on a phase by phase basis. Specifically,

$$CT = \sum_{p \in phases} max\{c_p^1, c_p^2, ..., c_p^m\}$$

where *phases* includes all instances of both compute phases and comphases, m is the number of processors, and c_p^j is the completion time of phase p on processor j.

The phase expressions drives the calculation of completion time in a manner similar to the calculation of communication volume described above. The precise computation of the completion time of each phase on each processor takes into account the multiplexing of processes and the overhead of message passing including startup time, dilation, and contention. In addition, we presume implicit barrier synchronization between phases in the computation of completion time. Details of this computation are beyond the scope of this paper, but are discussed in [LRG$^+$90b] and [SB90].

5.5 LaRCS provides an Efficient Representation of Task Graphs

The primary purpose of LaRCS is to describe the computation graph for the purpose of mapping. LaRCS' orientation towards the representation of *regular* parallel computations enables it to efficiently describe *families* of computation graphs. Thus, the size of the LaRCS representation of a parallel computation can be constant, i.e., independent of the number of processes (nodes) and communication edges. The LaRCS compiler translates user provided LaRCS code into intermediate code (an abstract syntax tree.) The LaRCS intermediate code can then be translated into the form needed for specific OREGAMI mapping algorithms such as the static task graph for the whole computation, the static task graph for a single `comphase`, or a DAG (TCG). OREGAMI provides a set of utility functions to generate specific data structures for these graphs, such as an adjacency matrix or DAG, given the parameters that instantiate the size of the problem instance. As mentioned earlier, LaRCS can also used to (laboriously) describe an arbitrary irregular task graph/adjacency matrix by declaring a unique `comtype` for each edge in the graph.

6 Conclusions and Future Work

We have presented a language for describing the communication patterns of *regular* parallel computations for the purpose of mapping. The key contributions of LaRCS are

- Its ability to describe regular communication topology and temporal communication behavior in a natural, compact notation.
- The utility of LaRCS in the context of OREGAMI for achieving efficient mappings.
- The underlying TCG model which enriches the static task graph model with temporal information, thereby increasing the range of scheduling and mapping algorithms that can be utilized.

LaRCS is capable of describing a wide range of parallel computations whose underlying abstraction is a collection of communicating parallel processes. At this time, LaRCS models deterministic computations whose communication topology and communication phases are known at compile time.

Our work with LaRCS is continuing in the following areas:

- **LaRCS notation for dynamically spawned tasks**

  We plan to expand LaRCS to include notation for describing computations which dynamically spawn tasks in a regular predictable pattern. Currently, the LaRCS

nodetype and **compute** declarations define static sets of nodes and their labellings. In order to describe dynamically spawned tasks, these need to be reinterpreted as *potential* sets of nodes and their labelings. A new construct needs to be introduced to identify parents and their spawned children. The phase expression could be augmented to describe the birth and death of these potential sets of nodes over time.

- **LaRCS-like notation for mapping**

  The current use of LaRCS is to serve as a compact notation for the communication structure and behavior of parallel algorithms for the specific goal of mapping. We also see the potential for developing a LaRCS-like notation to express the mapping itself. Since our focus is on the mapping of regular computations to regular interconnection networks, in many cases the mapping itself is regular and can be expressed in compact, parameterized notation. For example, our group theoretic contraction of the *n-body algorithm* to p processors could be expressed as:

  ```
  for i in 0..n CONTRACT(i) = i mod p
  ```

  There is also a need for notation to express constraints on portions of the mapping, such as assignment of certain processes to those processors tied to I/O devices or with specialized hardware. For example, given a mapping of a full binary tree with $n^2 - 1$ nodes to an $n \times n$ mesh (n^2 a power of 2), suppose the leaf nodes must be assigned to processors on the boundary of the mesh for purposes of I/O. This constraint could be expressed as:

  ```
  # This constrains mapping of leaves to north and south
  # boundary nodes of the mesh.  We presume nodetypes
  # treenode and meshnode have been defined for the
  # computation and interconnection network, respectively.

  for i in n**2/2 to n**2-1
          treenode(i) MAPSTO  meshnode(1,*) OR  meshnode(*,n);
  ```

- **LaRCS notation for data parallelism**

We would also like to use LaRCS for the closely related problem of data parallelism (data partitioning). Here, the problem is to express the assignment of portions of a shared data structure such as a matrix to individual processors, as well as the portions of the structure that need to be exchanged during execution of the parallel algorithm. LaRCS could be applied to this problem by (1) elaborating the `nodetype` declaration to describe the portions of the data structure assigned to each node as a function of the corresponding node label, and (2) augmenting the `comtype` declaration to include a description of the portion of the matrix transferred in that communication phase.

7 Acknowledgments

Thanks to Todd Mowry of MIPS and the anonymous referees for their helpful comments.

8 Bibliography

References

[AK89] S. B. Akers and B. Krishnamurthy. A group-theoretic model for symmetric interconnection networks. *IEEE Transactions on Computers*, C-38(4):555–566, April 1989.

[BC87] D. A. Bailey and J. E. Cuny. Graph grammar based specification of interconnection structures for massively parallel computation. In *Proceedings of the Third International Workshop on Graph Grammars*, pages 73–85, 1987.

[BC89] D. A. Bailey and J. E. Cuny. Visual extensions to parallel programming languages. In *Languages and Compilers for Parallel Computing*, pages 17–36, August 1989.

[Ber83] F. Berman. Edge grammars and parallel computataion. In *Proceedings of the 1983 Allerton Conference, Urbana, Il*, ?? 1983.

[Bok87] S. H. Bokhari. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishers, 1987.

[Bro86] J.C. Browne. Framework for fomulation and analysis of parallel computation structures. *Parallel Computing*, 3:1–9, 1986.

[BS84] F. Berman and G. Shannon. Edge grammars: Decidability results and formal language issues. In *Proceedings of the 1984 Allerton Conference, Urbana, Il*, ?? 1984.

[BS87] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, 4(5):439–458, October 1987.

[BS89] F. Berman and B. Stramm. Prep-p: Evolution and overview. Technical Report CS89-158, University of California at San Diego, 1989.

[ea90] S. Borkar et. al. Supporting systolic and memory communication in iwarp. In *Proceedings International Symposium on Computer Architecture*, 1990.

[ERL90] H. El-Rewini and T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[KK88] Simon M. Kaplan and Gail E. Kaiser. Garp: Graph abstractions for concurrent programming. In H. Ganzinger, editor, *European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 191–205, Heidelberg, March 1988. Springer-Verlag.

[Lo88] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, 37(11):1384–1397, November 1988.

[Lo91] V. M. Lo. Temporal communication graphs: A new graph theoretic model for mapping and scheduling in distributed memory systems. In *Proceedings 6th Distributed Memory Computing Conference*, April 1991.

[LRG$^+$90a] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. A. Mohamed, and J. Telle. Mapping divide-and-conquer algorithms ato parallel architectures. In *Proceedings IEEE 1990 International Conference on Parallel Processing*, pages III:128–135, August 1990.

[LRG$^+$90b] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. A. Mohamed, and J. Telle. OREGAMI: Software tools for mapping parallel algorithms to parallel architectures. In *Proceedings IEEE 1990 International Conference on Parallel Processing*, pages II:88–92, August 1990.

[MKS89] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, SE-15(6):663–675, June 1989.

[Pol88] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[SB90] B. Stramm and F. Berman. How good is good? Technical Report CS90-169, University of California at San Diego, Dept. of Computer Science and Engineering, 1990.

[Sei85] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[SER90] P. Sadayappan, F. Ercal, and J. Ramanujam. Clustering partitioning approaches to mapping parallel programs onto a hypercube. *Parallel Computing*, 13:1–16, 1990.

[Sny82] L. Snyder. Introduction to the configurable, highly parallel computer. *Computer*, 15(1):47–56, January 1982.

[Sto77] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[Vui87] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1987.

9 Appendix: LaRCS Grammar

LaRCS Program → *function definition* |
→ *attribute declaration* |
→ *label declaration* |
→ *node types* |
→ *communication types* |
→ *communication phases* |
→ *computation phases* |
→ *phase expression*

attribute declaration → attributes *name list*; |
→ λ

label declaration → nodelabels *range expression*..*range expression*; |
→ λ

node types → *node types node type* |
→ λ

communication types → *communication types communication type* |
→ λ

communication phases → *communication phases communication phase* |
→ λ

computation phases → *computation phases computation phase* |
→ λ

node type → nodetype *function definition label statement*

label statement → labels *range expression..range expression*; |
→ λ

communication type → comtype *function definition edge expression volume expression*

edge expression → *range expression* ⇒ *range expression*; |
→ *range expression* ⇔ *range expression*;

volume expression → volume = *expression*; |
→ λ

volume expression2 → volume = *expression*;

communication phase → comphase *function definition communication statements*

communication statements → *communication statements communication statement*
→ *communication statement*

communication statement → forall NAME in *expression..expression* {*communication statem*
→ for NAME = *expression* to *expression* {*communication staten*
→ if *expression* then *communication statement*
→ {*communication statements*}
→ *edge expression*
→ *function call*;

computation phase → compute *function definition computation statements*

computation statements → *computation statements computation statement*
→ *computation statement*

computation statement → forall NAME in *expression..expression* {*computation statement*
→ if *expression* then *computation statement*
→ *parameterized function call volume expression*2

phase expression → phase_expr *phase statement*; |
→ λ

phase statement → *phase statement* || *phase statement* |
→ *phase statement* |> *phase statement* |
→ *phase statement* ** *expression* |
→ forall NAME in *expression..expression* {*phase statement*} |
→ for NAME = *expression* to *expression* {*phase statement*} |
→ if *expression* then *phase statement* |
→ {*phase statement*} |
→ *function call*; |
→ compute

name → *name list* |
→ λ

name list → NAME |
→ *name list* , NAME

exp → *exp list* |
→ λ
exp list → *expression* |
→ *exp list* , *expression*

function definition → NAME |
→ NAME (*name*)

$$\begin{array}{rcll} function\ call & \rightarrow & \mathrm{NAME} & | \\ & \rightarrow & \mathrm{NAME}\,(exp) & \end{array}$$

$$\begin{array}{rcl} parameterized\ function\ call & \rightarrow & \mathrm{NAME}\,(exp) \end{array}$$

$$\begin{array}{rcll} range\ expression & \rightarrow & [exp\ list] & | \\ & \rightarrow & expression & \end{array}$$

$$
\begin{array}{lcll}
expression & \rightarrow & expression \; {*}{*} \; expression & | \\
 & \rightarrow & expression * expression & | \\
 & \rightarrow & expression/expression & | \\
 & \rightarrow & expression \; \% \; expression & | \\
 & \rightarrow & expression + expression & | \\
 & \rightarrow & expression - expression & | \\
 & \rightarrow & expression \; \& \; expression & | \\
 & \rightarrow & expression \mid expression & | \\
 & \rightarrow & expression{}^\wedge expression & | \\
 & \rightarrow & expression << expression & | \\
 & \rightarrow & expression >> expression & | \\
 & \rightarrow & expression < expression & | \\
 & \rightarrow & expression > expression & | \\
 & \rightarrow & expression >= expression & | \\
 & \rightarrow & expression <= expression & | \\
 & \rightarrow & expression == expression & | \\
 & \rightarrow & expression \; != expression & | \\
 & \rightarrow & expression \; \&\& \; expression & | \\
 & \rightarrow & expression \; || \; expression & | \\
 & \rightarrow & !expression & | \\
 & \rightarrow & +expression & | \\
 & \rightarrow & \text{-}expression & | \\
 & \rightarrow & (expression) & | \\
 & \rightarrow & function\ call & | \\
 & \rightarrow & \mathrm{NUMBER} &
\end{array}
$$