Temporal Communication Graphs: Lamport's Process-Time Graphs Augmented for the Purpose of Mapping and Scheduling

Virginia M. Lo

CIS-TR-92-05
January 1992

Department of Computer and Information Science
University of Oregon

Temporal Communication Graphs: Lamport's Process-Time Graphs Augmented for the Purpose of Mapping and Scheduling*

Virginia M. Lo
Dept. of Computer Science
University of Oregon
Eugene, Oregon 97403-1202
email: lo@cs.uoregon.edu

Abstract

The temporal communication graph is a new graph theoretic model of parallel computation that we have developed for the mapping of parallel computations to message-passing parallel architectures. The TCG can be viewed as an augmented version of Lamport's process-time graphs. It integrates the two dominant models currently in use in the areas of mapping and scheduling: the static task graph and the DAG. In addition, the TCG provides the capability to identify logically synchronous *phases* of communication and computation, and to describe the temporal behavior of a parallel algorithm in terms of these phases. This paper defines the TCG, introduces a language for describing TCGs, illustrates its use for mapping and scheduling, and discusses a wide range of potential uses for the TCG in the area of parallel programming environments.

*This research was sponsored by NSF grant CCR-8808532 and the Oregon Advanced Computing Institute (OACIS)

1 Introduction

The temporal communication graph (TCG) is a new graph theoretic model of parallel computation that we have developed for the mapping of parallel computations to message-passing parallel architectures. The TCG can be viewed as a Lamport process-time graph with augmentations for the purpose of mapping and scheduling. Both the TCG and its forerunner, the classic static task graph of [25], are designed for systems in which the programmer designs his or her program as a set of communicating parallel processes which use explicit message-passing. The TCG was designed to enrich the static task graph model (1) by adding temporal information to the model and (2) by providing a means for describing regularity in the structure of the parallel computation. These features of the TCG were motivated by our observations that many parallel algorithms are characterized by both topological and temporal regularity, and that regularity can effectively be exploited for mapping and scheduling.

- **The integrative nature of the TCG enables a wide spectrum of algorithms to be utilized for mapping and scheduling which could not otherwise be invoked because of incompatibilities in the underlying graph theoretic models.** The TCG models the static task graph and also models 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 processes comprising the computation. Thus, the TCG is a hybrid of the two dominant models currently in use in the areas of mapping, task assignment, partitioning, and scheduling: the static task graph model and the precedence-constrained DAG. Because the TCG is compatible with both of these important models, algorithms for static task assignment and mapping such as [25], [5], [3], [15], [21], and scheduling algorithms for DAGS such as [6], [22], [27], [8], can be applied to our model.

- **The ability of the TCG to capture regularity enables the development and use of specialized algorithms which exploit the regularity to yield improved performance.** We have observed that many computations consist of one or more distinct *communication phases* characterized by a *regular* topological structure. In addition, the behavior of the parallel computation can often be expressed as a regular pattern of these communication phases over time. The TCG and our graph description language LaRCS provide a means for defining these phases and expressing the temporal patterns in a notation called *phase expressions*. The OREGAMI [16]

mapping algorithms utilize information about regularity to perform task contraction, placement, and routing.

- **The generality of the TCG, particularly its relationship to Lamport's process-time graphs, enables it to serve as a unified model of parallel computation in parallel programming environments.** In addition to its uses for mapping and scheduling, the TCG model has the potential to be used for program development, debugging, and performance monitoring.

In this paper, we define the TCG and its relation to Lamport's model. We describe our graph description language LaRCS, which provides a natural, parameterized means for efficiently describing the TCG. We discuss in detail the use of the TCG for mapping and scheduling. In addition, we show how the generality of the TCG makes it potentially useful as an abstraction in parallel programming environments. We summarize by evaluating the TCG with respect to expressive power, utility, efficiency, and performance.

2 The Temporal Communication Graph

In this section we first describe how a TCG models a parallel computation, including its spatial and temporal structure, as well as regular patterns in the structure of the computation. We then give a formal definition of the TCG as a graph theoretic model of parallel computation.

2.1 Related models of parallel computation

Before describing the TCG, we briefly present the two graph theoretic models of parallel computation that are most widely used for mapping and scheduling: the static task graph of [25] and the precedence constrained DAG model. We also describe Lamport's process time graph which is used to address issues of time, synchronization, and coherence in distributed systems. The contribution of the TCG is that it unifies these three important models, providing a rich foundation for use in parallel programming environments, particularly scheduling and mapping.

The static task graph was designed to model a set of communicating parallel processes which use explicit message passing. Each process is represented as a node, and edges represent the fact that one or more messages are sent from the sending process to the receiving process sometime during the lifetime of the computation. Weights on nodes and edges represent the cumulative execution costs and interprocessor communication costs, respectively, occurring during the lifetime of the computation.

In the DAG model of parallel computation, nodes represent computational units (tasks) while directed edges represent the precedence relationships among these tasks. Weights on nodes represent task execution costs while weights on edges represent the interprocessor communication cost incurred before the successor task can be executed.

In Lamport's model of parallel computation, nodes represent one of three atomic events: computation, sending a message, or receiving a message. All events associated with a given process form a linear chain of nodes with directed edges indicating the ordering of the events. Directed edges from a send event to a receive event represent message-passing. Lamport's model does not include weights on nodes or edges.

Other models of parallel computation which are not directly utilized in our work include the dataflow model, the actors model, and communicating finite automata and its offshoots.

2.2 Description of the TCG

As we mentioned earlier, the TCG was designed to model parallel computations which have been designed by the programmer as a set of communicating parallel processes that utilize explicit message-passing. The activity of a given process p_i can be seen as a sequence of atomic events, where each event is either a computation event or a communication event: sending a message or receiving a message. In the TCG, the sequence of atomic events in process p_i is represented as a linear chain of nodes, with directed edges indicating the precedence relationship between the events. A message sent from process p_i to process p_j is represented with a directed edge from the send-event node on p_i to the corresponding receive-event node on p_j.

Note that the TCG as we have described it up to this point is identical to Lamport's process-time diagrams [13]. We augment Lamport's model with weights associated with the nodes and edges, used to represent computation and communication costs, respectively. In addition, a mapping function on subgraphs of the TCG is used to identify computation or communication *phases*: sets of *logically synchronous* compute events or communication events, respectively. These *phases* are defined and explained more fully in the next subsection.

The TCG can be seen as an unrolling of the static task graph over time to form the Lamport process-time graph. Each node of the static task graph which represents one process, p_i, is unrolled into a linear chain of nodes representing the compute, send, and receive events occurring during p_i's lifetime. Conversely, the projection of the TCG along the time axis yields the static task graph. In the remainder

of this section, we shall distinguish between the TCG-DAG and the static-TCG when referring to the corresponding graphs. The TCG-DAG corresponds to the Lamport process-time representation of the computation while the static-TCG corresponds to the projection of the TCG-DAG. We shall use the term TCG when we wish to refer to the TCG model as a whole.

We illustrate the static-TCG and the TCG-DAG using a parallel algorithm for the *n-body* problem which was designed for the Cosmic Cube [23]. Figure 1 shows the static-TCG for the *n-body* algorithm and describes its operation. This computation is characterized by two distinct compute phases and two distinct communication phases. Figure 2 shows the TCG-DAG in which each process of the static task graph is unrolled in time.

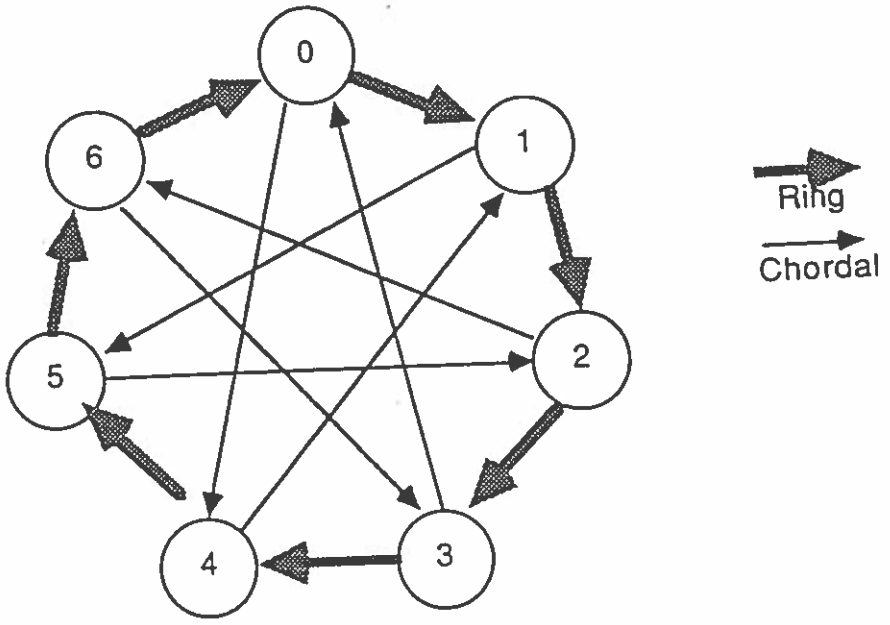
2.3 Describing Regularity in the TCG

We note that many practical parallel algorithms involve one or more *phases* of computation and communication. Communication phases are often characterized by regularity in the communication topology, such as a mesh or tree topology. In addition, these computations also exhibit regularity in the temporal behavior, i.e., the patterns of the phases that are active over time.

Thus, in addition to identifying the atomic compute and communication events, the TCG also allows the programmer to identify sets of atomic events that are spatially or temporally regular. A *compute phase* corresponds to a set of nodes in the TCG-DAG (compute events) that are involved in *logically synchronous* computation. A *communication phase* corresponds to a set of edges (sender/receiver pairs) in the TCG-DAG that are involved in *logically synchronous* communication[1]. By *logically synchronous* we mean that at run time 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.

Because a given phase may occur more than once in a parallel computation, it is necessary to distinguish between *phase types* and *phase occurrences.* A phase type is a template describing the structure of a phase in the computation. A phase occurrence is an instantiation of a specific phase type within the TCG-DAG. As we shall see in Section 3, the LaRCS description language provides mechanisms

[1]Not to be confused with the notion of *synchronous message-passing in which sender and receiver must be ready to communicate before message-passing can occur [1].*



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.

Figure 1: Task graph for the *7-body algorithm*

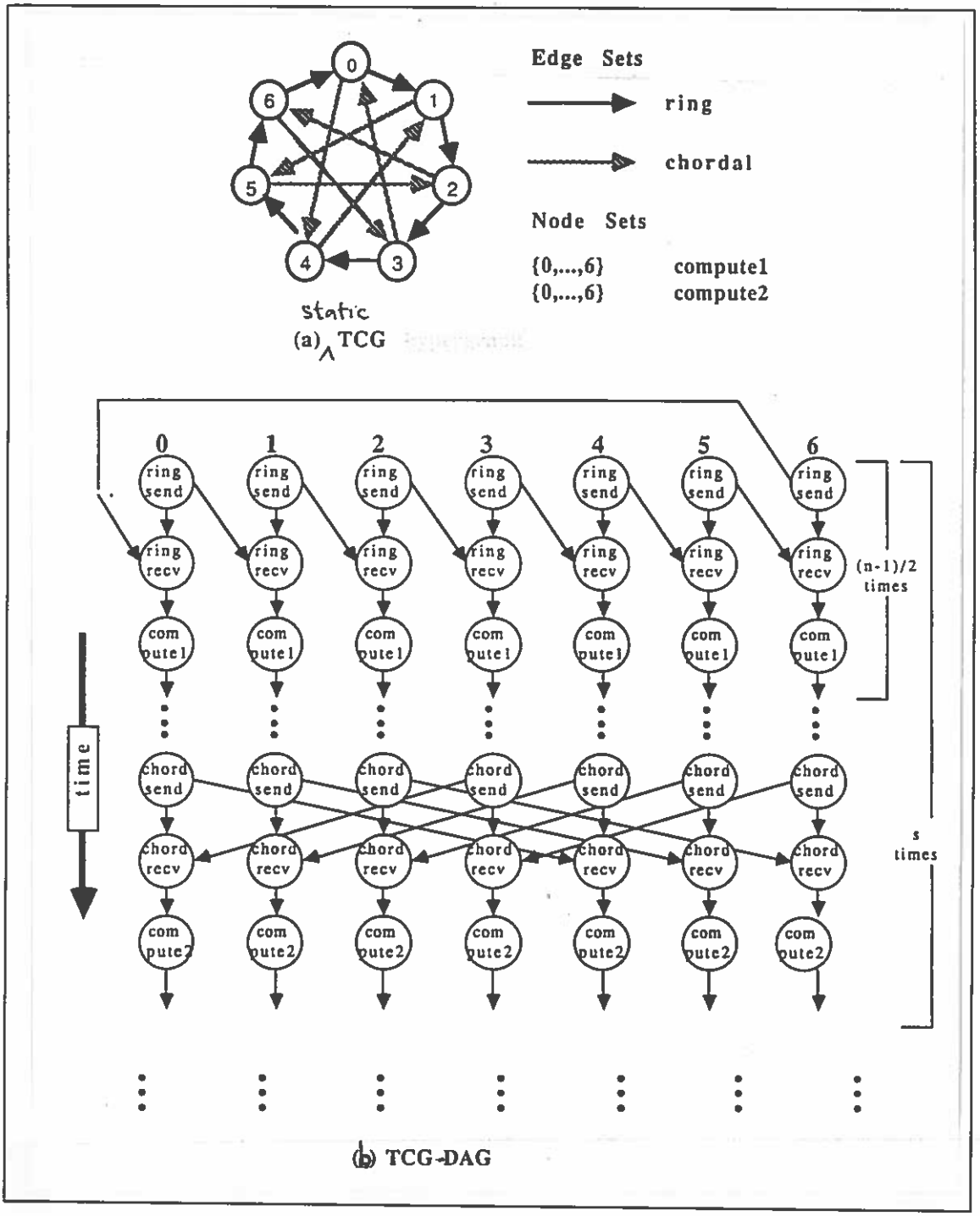


Figure 2: TCG-DAG for the *n-body* algorithm

for defining a phase type using parameterized communication functions and for instantiating phase occurrences using a notation called *phase expressions.*

2.4 Formal Definition of the TCG

The definition of the TCG involves three parts: the static-TCG, the TCG-DAG, and a collection of small graphs called phase-type DAGS. The static-TCG is a task graph in which nodes correspond to processes and directed edges to communication and is identical to the static task graph of Stone [25]. One or more subsets of nodes and one or more subsets of edges identify the compute phase types and the communication phase types, respectively. Associated with each subset is a DAG called the phase-type-DAG which which serves as a template for instantiation of phase occurrences in the TCG-DAG. The TCG-DAG is a Lamport process-time graph, in which nodes correspond to atomic events (compute events, send events, receive events), and edges represent precedence or message-passing. The TCG-DAG can be partitioned into disjoint subgraphs, where each subgraph in the TCG-DAG is a *phase occurrence* which is isomorphic to a unique phase-type-DAG.

Definition of the static-TCG: Let $P = \{P_0, P_2, ...P_{n-1}\}$ be a parallel computation consisting of n communicating processes numbered 0 through $n-1$. The static-TCG is a weighted directed graph $G = (V, E)$, with $|V| = n$. Each $v_i \in V$ represents the process p_i and each edge $e_{ij} \in E$ represents the cummulative message-passing from process p_i to process p_j during the lifetime of the computation. A labeling function L labels each node v_i with its process id i. A weight function W assigns each of the nodes and edges in G a weight which represents the cumulative execution and communication costs, respectively.

Definition of the phase-types: Let $\hat{V}$ be a family of subsets of V and let $\hat{E}$ be a family of subsets of E. Each V_t in $\hat{V}$ identifies a subset of the nodes of V and corresponds to a compute phase type. Each E_s in $\hat{E}$ identifies a subset of the edges of E and corresponds to a commmunication phase type. We constrain E_s such that no two edges may have the same sending node and no two edges may have the same receiving node[2] A labeling function $\hat{L}$ labels nodes in V_t and the send nodes and receive nodes of edges in E_s with the ids of the corresponding processes. A weight function $\hat{W}$ assigns weights to the nodes of V_t which

[2]This constraint is needed to conform to the structure of Lamport process-time graphs. It has implications for certain types of communication such as *multicast* which are discussed in Section 2.5.

represent the execution time of that node, during ONE occurrence of the corresponding compute phase type. $\hat{W}$ assigns weights to the edges of E_s which represent the message volume on that edge, during ONE occurrence of the corresponding communication phase type.

Definition of the phase-type-DAGs: A constructor $\delta(V_t)$ builds a DAG called the phase-type-DAG for each compute phase type $V_t \in \hat{V}$. The phase-DAG consists of the (unconnected) set of $|V_t|$ nodes, labeled and weighted by $\hat{L}$ and $\hat{W}$, respectively.

A constructor $\delta(E_s)$ builds a DAG called the phase-DAG for each communication phase type $E_s \in \hat{E}$. Each edge e_{ij} in E_s is one of the edges of the static-TCG G and represents a message sent from process p_i to processp_j. For each e_{ij} we construct two nodes n_i and n_j to represent the send-event and receive-event, respectively, and a directed edge from n_i to n_j. The nodes are labeled with their process ids according to $\hat{L}$. The edge (n_i, n_j) is given weight $\hat{W}(e_{ij})$. After this construction is completed for all e_{ij} in E_s, we check to see if there exists any pairs of send-nodes and receive-nodes with identical labels (process ids). If so, a precedence edge is constructed from the send-node to the receive-node.

The *n-body algorithm* has two compute phase-DAGs and two communication phase-DAGs. These are shown in Figure 3.

Definition of the TCG-DAG: The TCG-DAG is a directed acyclic graph $G' = (V', E')$ and a labeling L' of the nodes of V'. The nodes of V' correspond to the compute events, send-events, and receive-events occurring in the parallel computation. L' labels each $v' \in V'$ with the id of the process on which it occurs. $E' = E_P \bigcup E_M$ where E_P are precedence edges that give a total ordering to the events whose process ids are identical, and E_M are message passing edges and therefore constrained to be incident to nodes whose process ids are different. There is at most one incoming and at most one outgoing edge of E_P incident to each node. There is at most one incoming OR one outgoing message edge of E_M incident to each node.

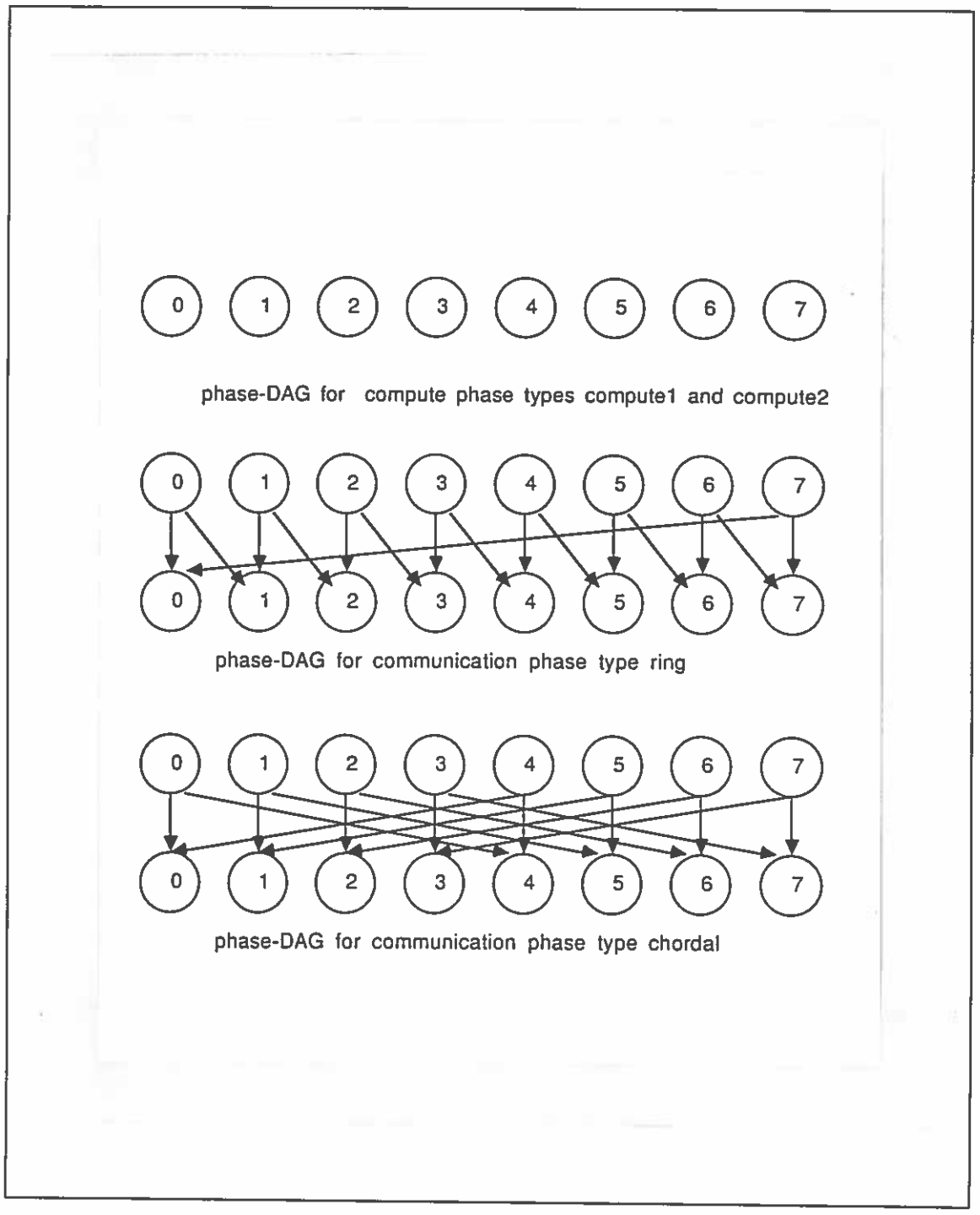


Figure 3: Phase-DAGs for *nbody algorithm* compute and communication phase types

Constraints on the relationships between the static-TCG, the phase-DAGs, and the TCG-DAG:

- The TCG-DAG G' can be partitioned into disjoint subgraphs such that each subgraph is isomorphic to one of the phase-DAGs $\delta(E_s)$ or $\delta(V_s)$. This condition means that the TCG-DAG consists of a set of phase occurrences instantiated from the phase types.
- The weight function W satisfies the constraints that for any node (edge) in G the weight on that node (edge) equals the sum of the weights assigned by $\hat{W}$ to all of the corresponding nodes (edges) in all phase occurrences in the TCG-DAG

2.5 Limitations of the TCG

Because the atomic communication events of the TCG and the Lamport process-time graph are single send or single receive events, neither of these models represents broadcast (multicast) communication as simultaneous events. In the TCG, broadcast (multicast) must be represented as a sequence of messages from the sender to each of the receivers. We note, however, that the LaRCS `comphase` declaration will cause mapping algorithms to view the sequence of messages as a simultaneous broadcast (multicast). In addition, the TCG abstraction is not designed to model conditionals or non-determinism.

3 LaRCS: a language for describing TCGs

LaRCS is a description language we have developed [16], [17] which describes the TCG in an efficient and intuitive notation. 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 can be constant, i.e., independent of the number of process nodes and communication edges in the TCG. The LaRCS compiler translates the user-provided LaRCS code into intermediate code in the form of an abstract syntax tree. This tree is used by OREGAMI utility functions to generate the specific representation of the TCG needed for the mapping algorithms: static-TCG, TCG-DAG, or phase-type-DAGS.

The version of LaRCS described below does not describe all possible temporal communication graphs (and Lamport process-time graphs.) We believe that it does describe a large enough class of graphs to cover many practical parallel programs. LaRCS has been tested on a broad spectrum of parallel computations 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. An extension to LaRCS which describes all possible TCGs (and Lamport process-time graphs) is currently under development.

3.1 LaRCS Language Features

A LaRCS program consists of the following major components: (a) the LaRCS `nodetype` declaration which describes the processes, (b) the `computephase` declarations which define the compute phases types, (c) the LaRCS `comtype` and `comphase` declarations define the communication phase types and (d) the LaRCS `phase expression` which instantiates all the edges of a parallel computation using the `comtype` and `comphase` declarations. A `comtype` declaration describes a single communication edge; a `comphase` describes a set of logically synchronous communication edges. A `phase expression` describes the entire parallel program in terms of its temporal computation and communication behavior.

Fig. 4 gives the LaRCS code for the *n-body* algorithm. The line numbers in the LaRCS code are used for reference purposes only in the following commentary.

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[3].

2. **Attributes.** The programmer may specify global characteristics of the task graph, such as `nodesymmetric` or `planar`. This information is used to select appropriate mapping algorithms.

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 multi-dimensional 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 and a default labeling is used.

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

```
1.    nbody(n,s)
2.    attributes nodesymmetric;
3.    nodetype body
         labels 0..(n-1);
4.    computephase compute1
         forall i in 0..(n-1) body(i); volume = COST1;
      computephase compute2
         forall i in 0..(n-1) body(i); volume = COST2;
5.    comtype ring_edge(i) body(i) => body((i+1) mod n);
         volume = MSGSIZE;
      comtype chordal_edge(i) body(i) => body((i+(n+1)/2) mod n);
         volume = MSGSIZE;
6.    comphase ring
         forall i in 0..(n-1) {ring_edge(i);}
      comphase chordal
         forall i in 0..(n-1) {chordal_edge(i);}
7.    phase_expr
         {{ring |> compute1}**(n-1)/2 |> chordal |> compute2}**s;
```

Figure 4: LaRCS code for the n-body algorithm

4. **Computephase declaration.** A `computephase` declaration defines a compute phase type. In this example, by default both compute phase types, `compute1` and `compute2` involve all nodes of type `body`. The `volume` parameter specifies the execution cost for one occurrence of the corresponding phase type.

5. **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. In Fig. 4, there are two `comtype` declarations: `ring_edge` and `chordal_edge`. The `volume` field of the `comtype` declaration is an arithmetic expression which specifies the message volume (typically in bytes) of a single message transfer.

6. **Comphase declaration.** A `comphase` defines a commnication phase type, 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. The `comphase` declaration is constrained to define a set of edges such that no two edges have the same sender and no two edges have the same receiver[4]. In the *n-body algorithm* there are two `comphase` declarations: `ring` and `chordal`.

7. **Phase expression.** The phase expression describes the temporal behavior of the computation in terms of its compute and communication phases types[5]. The phase expression defines the TCG-DAG by instantiating occurrences of the phase types. Phase expressions are defined recursively below where r and s are phase expressions.

   - a single `computephase` is a phase expression.
   - 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 parameter in r.
   - parallelism: r `||` s is a phase expression denoting parallel execution of phases r and s.

[4]This is the constraint specified in the formal definition of the TCG.
[5]The seed for our idea of phase expressions originated with [18].

- 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*.
- span: *r* `[` *s* is a phase expression denoting asymmetric parallelism in which the events of *r* start before the events of *s* and complete after the completion of *s*[6].

Only expressions derived by a finite application of these rules are phase expressions. A precise definition of the semantics of the operators `|>` , `||` , and `[` are given in the next section. A full discussion of LaRCS is given in [17].

3.2 The Semantics of LaRCS: Construction of a TCG from the LaRCS code

In this section we define LaRCS semantics by describing the construction of the TCG-DAG from the LaRCS code. The construction of TCG-DAG T is driven by the phase expression P and is defined recursively below.

- If P is a `computephase` and V is the set of nodes specified by P, then T is the phase-DAG $\delta(V)$ defined in Section 2.4.
- If P is a `comphase` and E is the set of edges specified by P, then T is the phase-DAG $\delta(E)$ defined in Section 2.4.

Suppose P is of the form P_1 op P_2 where op $\in$ { `|>` , $\|$, `[` } . Let T_1 and T_2 be the TCGs constructed from the phase expressions P_1 and P_2, respectively. Then T can be constructed recursively from T_1 and T_2 as described below. We first define some terms and operations used in the construction. These definitions are illustrated in Figure 5.

- **Definition:** An *intra-process edge* in T is an edge from an event associated with a specific process to the next event associated with that same process. Note that by construction there will be at most one intra-process edge associated with any node. A *message edge* in T is an edge from a send event on one process to the corresponding receive event on a different process. Note that by construction there will be at most one message edge associated with any node. A node u *precedes* a node v if there is a directed path in T from u to v. Similarly, an edge e *precedes* a node v if there is a directed path in T whose first edge is e and whose last node is v.

[6]The span operator allows the TCG to include graphs in which messages are received out of order. We note that Lamport also relaxes the constraint that messages be received in the order sent.

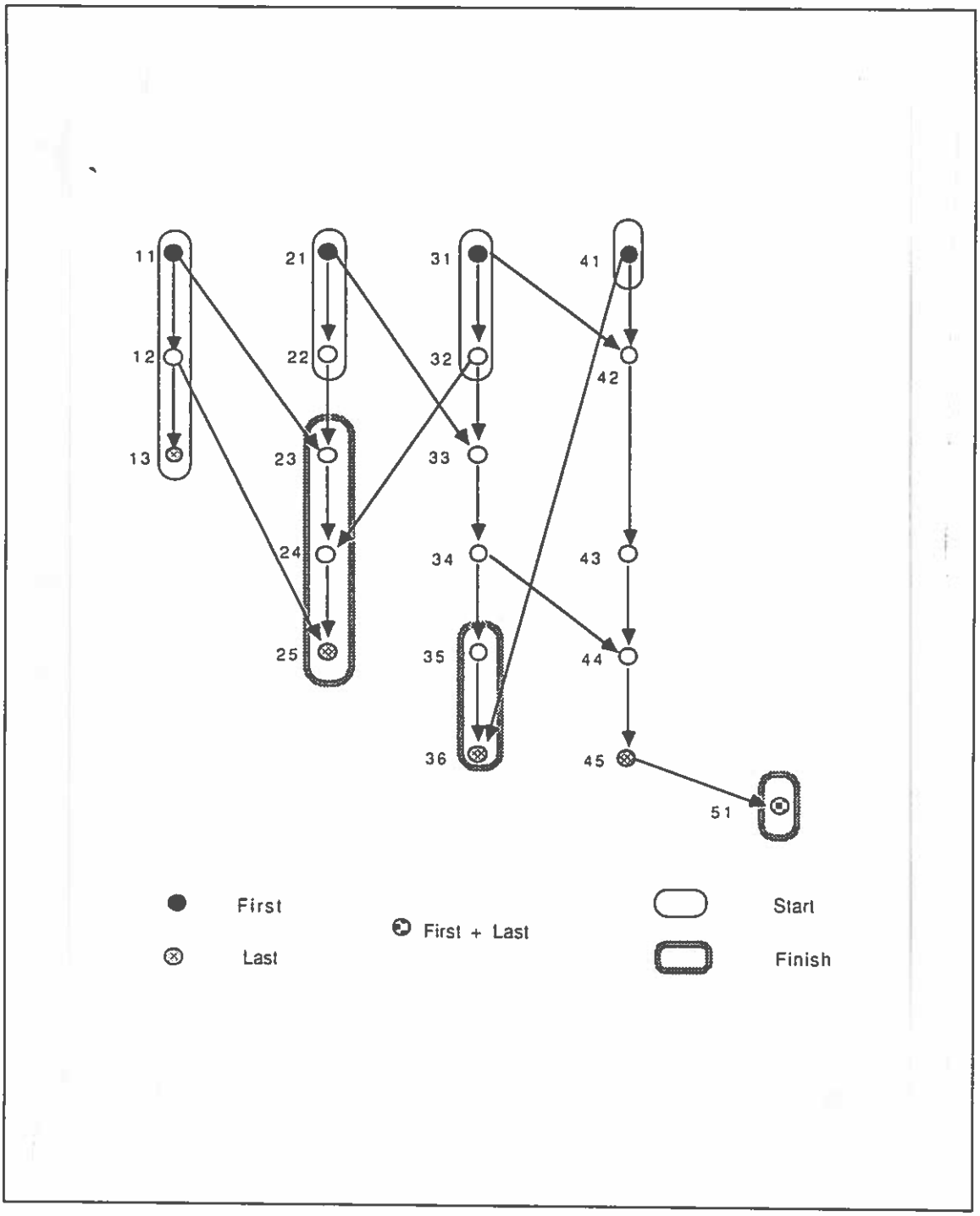


Figure 5: *start*, *finish*, *first*, *last*

Definition:

$start(T)$ are those nodes in T which are not *preceded* by any message edges.

$finish(T)$ are those nodes which are not in $start(T)$ and which do not precede any node with a different process id.

$middle(T) = nodes(T) - start(T) - finish(T)$.

Observation: $start(T)$, $middle(T)$, and $finish(T)$ form a partition of the nodes of T. There exists a Lamport logical clock function on T such that the logical times assigned to events in $start(T)$ are less than the logical times assigned to events in $middle(T)$ which are in turn less than the logical times assigned to events in $finish(T)$.

Definition:

$first(T)$ are those nodes in T which are not *preceded* by any intra-process edge.

$last(T)$ are those nodes in T which do not *precede* any node with the same process id.

Let PID_T be the set of process id's occurring in T. For each $i \in PID_T$ there is a unique node f_i^T in $first(T)$ and a unique node l_i^T in $last(T)$, possibly $f_i^T = l_i^T$.

Intuitively, *first* defines the union of the first events occuring on each of the processes; f_i^T is the first event occuring on process i. Similarly, *last* defines the union of the last events occuring on each of the processes; l_i^T is the last event occuring on process i.

- Let phase expression $P = P_1 \ |> \ P_2$. Then T is constructed as $T_1 \bigcup T_2$ with the addition of intra-process edges from node $l_i^{T_1}$ in $last(T_1)$ to node $f_i^{T_2}$ in $first(T_2)$, for all $i \in PID_{T_1} \bigcap PID_{T_2}$. Intuitively, this means intra-process edges are added from the last events of T_1 to the first events of T_2 whenever the events occur on the same process.

- Let phase expression $P = P_1 \ [\ P_2$. Then T is constructed as $T_1 \bigcup T_2$ with the addition of one intra-process edge from node $l_i^{start(T_1)}$ in $last(start(T_1))$ to node $f_i^{start(T_2)}$ in $first(start(T_2))$, and one intra-process edge from node $l_i^{finish(T_1)}$ in $last(finish(T_1))$ to node $f_i^{finish(T_2)}$ in $first(finish(T_2))$, for all $i \in PID_{T_1} \bigcap PID_{T_2}$. (See Figure 6.) Very roughly speaking, this means that intraprocess

edges are added from the starting events of T_1 to the starting events of T_2; and from the finishing events of T_2 to the finishing events of T_1.

We require that the following constraint must hold for T_1 and T_2:

$$PID_{middle(T_1)} \bigcap PID_{middle(T_2)} = \emptyset.$$

This constraint is necessary to ensure that there is no ambiguity in the ordering of events in $middle(T_1) \bigcup middle(T_2)$.

- Let phase expression $P = P_1 \parallel P_2$. Then T is constructed as $T_1 \bigcup T_2$ with the addition of one intra-process edge from node $l_i^{start(T_1)}$ in $last(start(T_1))$ to node $f_i^{finish(T_2)}$ in $first(finish(T_2))$, for all $i \in PID_{T_1} \bigcap PID_{T_2}$; and one intra-process edge from node $l_i^{start(T_2)}$ in $last(start(T_2))$ to node $f_i^{finish(T_1)}$ in $first(finish(T_1))$ for all $i \in PID_{T_1} \bigcap PID_{T_2}$. Note that parallel is the symmetric version of span.

  We require that the following constraint must hold for T_1 and T_2. The only sets which are allowed to have nonempty intersection are:

  $PID_{start(T_1)} \bigcap PID_{finish(T_2)}$ can be nonempty.
  $PID_{start(T_2)} \bigcap PID_{finish(T_1)}$ can be nonempty.

  This constraint is necessary to ensure that there is no ambiguity in the ordering of events in $T_1 \bigcup T_2$. For the two sets with nonempty intersection, the ordering is explicitly given.

- If P is sequential repetition or sequential loop, it can be written in the form $P = P_1$ |> P_2 |> ... |> P_k after instantiation of the parameters. In this case, we presume left to right precedence and construct the TCG according to the rules for |>.

- If P is parametrized parallel phase expression, it can be written in the form $P = P_1 \parallel P_2 ... \parallel P_k$ after instantiation of the parameters. In this case, we presume left to right precedence and construct the TCG according to the rules for ||.

Some simple phase expressions and the corresponding TCG-DAGs are shown in Figure 6.

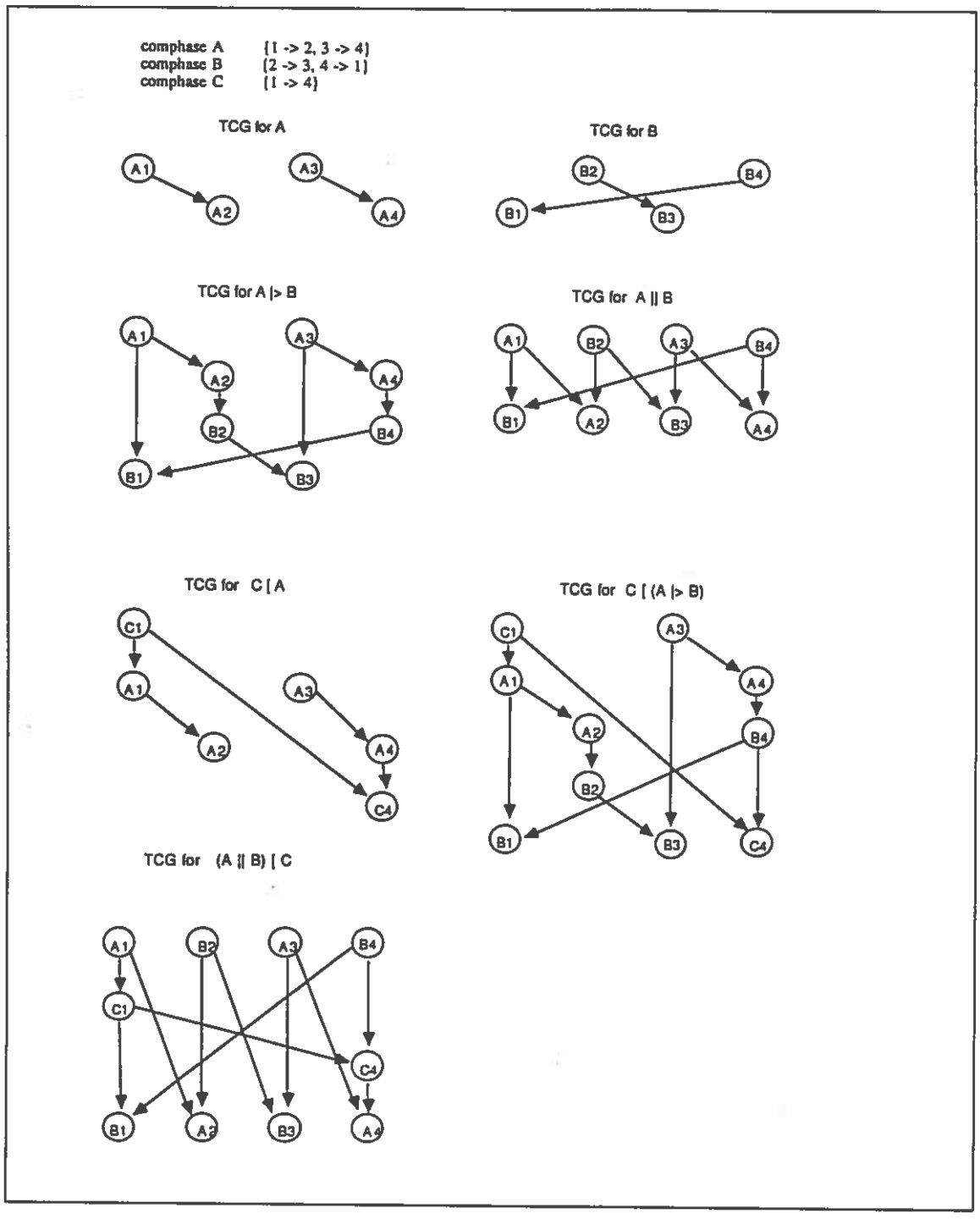


Figure 6: Some simple phase expressions and their TCGs (continued on next page)

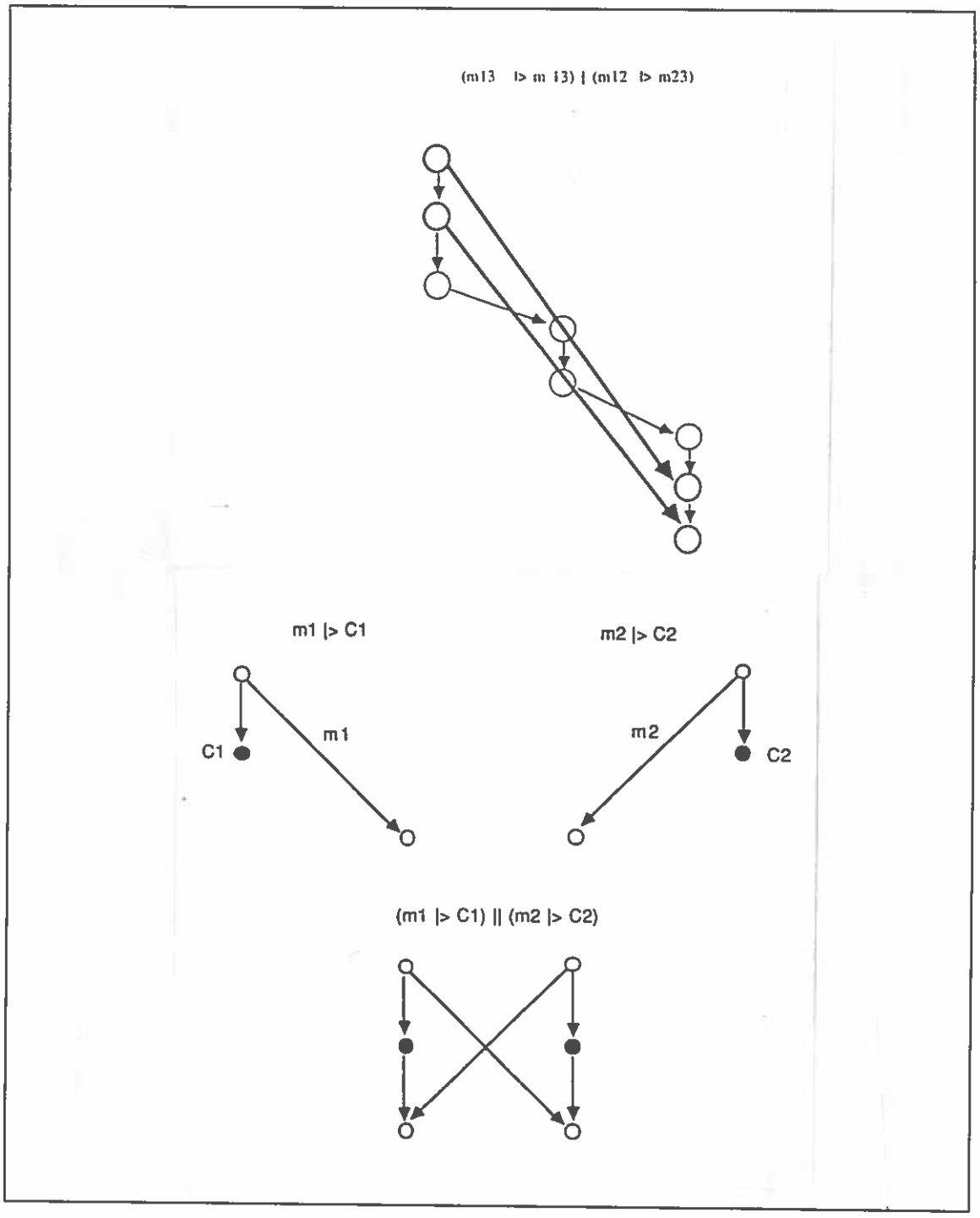


Figure 6: (continued) Some more simple phase expressions and their TCGs

4 Mapping and scheduling based on the TCG

As we indicated earlier, the TCG integrates the static task graph model of [26] and the precedence-constrained DAG model used for multiprocessor scheduling. Task assignment and scheduling research utilizing these two models has more or less followed disjoint paths, in that techniques and algorithms developed for one model have not been applicable to the other. In this section, we show how the TCG facilitates the use of algorithms from both domains and how the ability of the TCG to capture regularity enables specialized algorithms to be invoked which offer enhanced performance because they exploit regularity.

The static task graph has been widely used for the *task assignment* problem or *mapping problem* where the goal has been the assignment of the tasks (processes, nodes) to processors and the assignment of messages to links in order to minimize interprocessor communication and/or to achieve load balancing. A wealth of research has been conducted over the past 15 years utilizing this basic model including [26], [5], [3], [15], [22] to name just a few. Much of this research utilizes algorithms that are closely tied to the use of the static task graph model and which cannot be adapted to the DAG model. These include algorithms developed from graph theory such as network flow algorithms, matching algorithms, and graph embedding algorithms, edge grammars, and clustering techniques. The shortcomings of the static task graph model are its restriction to large-grained parallelism and its inability to capture temporal information. The latter constraint has limited the use of this model for the scheduling or multiplexing of processes.

The DAG model of parallel computation was developed for the parallelization of sequential code. The DAG representation is typically used to model smaller grained tasks and represents the data dependencies and precedence constraints among these tasks. The DAG has been used for the *scheduling problem* in which tasks are assigned to a processor and to a time slot. Researchers who have developed DAG-based algorithms include [7], [23], [6], [21], [14], [28], [20] and many more. These include the many list scheduling algorithms, critical path algorithms, and techniques such as linear clustering. The DAG model fails to capture the spatial topology of the computation when it is defined as a set of communicating parallel processes.

The TCG integrates these two models by representing both the static task graph and the DAG as an augmented Lamport process-time graph. The result is much more flexiblity in the mapping and scheduling of parallel computations. For example, algorithms for static task assignment can be applied

to the static-TCG to perform contraction, placement, and routing, followed by the invocation of a DAG-based algorithm to perform scheduling. Alternatively, a DAG-based algorithm can be used As stated earlier, the TCG was designed for systems in which the programmer designes his/her code as a set of communicating parallel processes. Thus, if DAG-based scheduling algorithms are applied to the TCG, it is necessary to ensure that the scheduler assign all of the subtasks comprising a single process to the same processor. Specifically, in order for a "vertical" chain of nodes to retain their identity as a single process (and thus ensure that they are all assigned to the same processor), each of the vertical edges can be given a weight of infinite cost.

In addition, the TCG can be used within programming environments which do preserve the identity of the individual processes. In this case, intraprocess precedence edges can be treated as message-passing edges: the weights on the edges correspond to the volume of state information that needs to be transferred for the successor events to occur. As a result, the scheduling algorithms will treat them as individual schedulable units and may assign them to different processors.

Figure 7 shows the use of the TCG model for a parallel algorithm for divide-and-conquer computations in three different cases. (i) In the first case, mapping algorithms developed for the static task graph model are applied to the static-TCG. The specific algorithms utilized here are the OREGAMI algorithms for contraction, placement, and routing which do not utilize regularity. (ii)The second case shows the application of the Mapping Heuristic (MH), an algorithm developed by [8] for the DAG-model. It is applied to a version of the TCG with infinite weights on the precedence edges. (iii) In the third case, we use OREGAMI algorithms which take advantage of the regular structure of the divide and conquer algorithm. The topology of the communication graph is a binomial tree and thus a canned mapping from the OREGAMI library is invoked for placement and routing [16]. Assuming unit execution and communication costs, the completion time for the four algorithms is 23, 28, and 21, respectively. The best performance is achieved by the OREGAMI canned mapping which uses knowledge of the regular task structure. In particular, the canned mapping assigns pairs of tasks from the same `computephase` to different processors and assigns messages from the same `comphase` to different links, thereby avoiding contention and maximizing load balancing. In addition, the canned mapping has optimal dilation of 1 hop. The OREGAMI static task graph mapping performed almost as well but because it did not have phase information, it assigned pairs of tasks that are logically synchronous to the same processor. In addition, the task assignments resulted in some of the messages being two hops away. Finally, the MH algorithm's greedy approach yielded unbalanced loads and possibly high communication overhead,

resulting in overall longer completion time.

In this section we also give a few brief examples to illustrate how the TCG's capability to identify compute phases and communication phases, and the temporal information provided by the phase-DAG can be profitably used by mapping algorithms. These methods are described in more detail in [18].

- OREGAMI includes a library of canned mappings for computations whose communication topology matches well-known graph families such as binary trees, binomial trees, rings, lines, hypercubes, meshes, tori, cube-connected cycles, butterflies, and FFT. Mapping for these structures to the hypercube, the mesh, and the deBruijn network topologies are retrieved from the OREGAMI library given the name and parameters of the computation graph and the network graph.

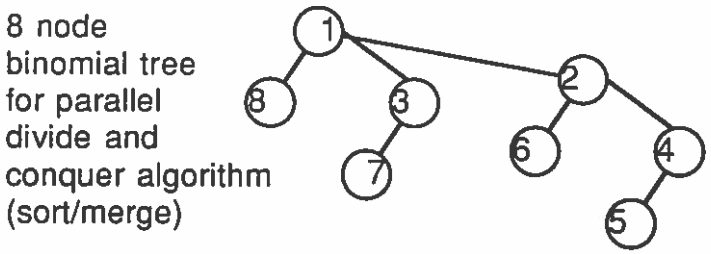
- OREGAMI includes a group theoretic algorithm for contracting certain highly-symmetrical task graphs, yielding a perfectly load-balanced and symmetric contraction whenever the task graph is a Cayley graph. This algorithm utilizes the LaRCS `comphase` declarations to aid in the recognition of the Cayley graph by viewing them as possible generators of an underlying group [27].

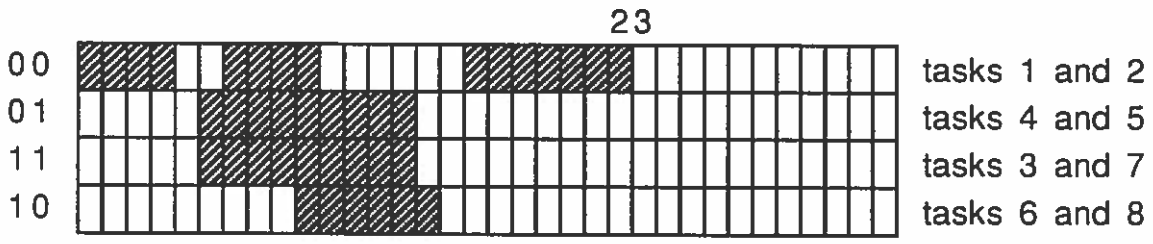
- All of the OREGAMI routing algorithms minimize contention by attempting to find a disjoint set of routes to service the set of messages occurring in a single `comphase`. This enables the routing algorithms to focus only on those messages that are capable of actual contention at runtime. The likelihood of finding a low contention routing is greater because fewer communication edges are considered; and the use of contention as a performance metric is more accurate since we avoid measuring *false contention* which occurs when two edges which are not active simultaneously at run time are mapped to the same link. We note that our routing algorithms which utilize information provided by the TCG have significantly less contention than fixed routing schemes such as *ecube routing* for the hypercube and *XY routing* for the mesh [30].

- The LaRCS phase-expression is used to calculate cumulative computation and communication costs based on single node and edge costs and to guide estimation of the total completion time of the parallel computation.

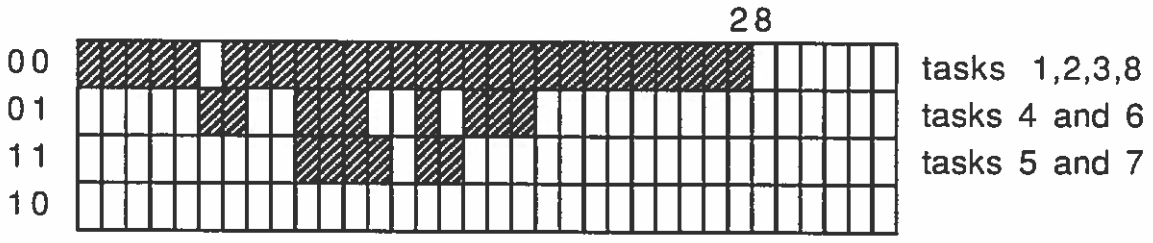
5 The use of the TCG in Parallel Programming Environments

We believe that the TCG model has the potential to serve as a unifying abstraction throughout the domains encompassed by parallel programming environments including program development, mapping

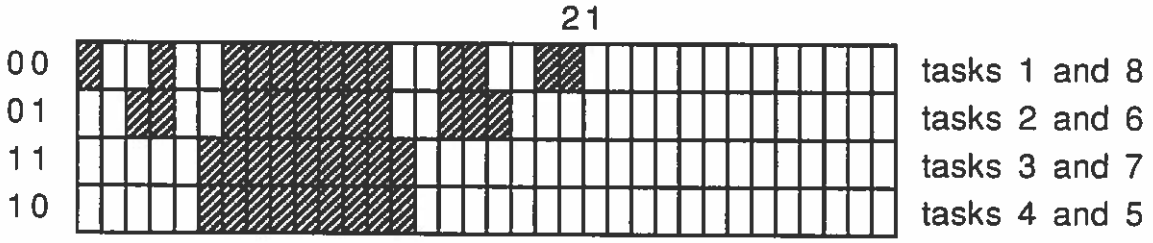




(i) OREGAMI default mapping using static-TCG



(ii) MH mapping using TCG-DAG



(iii) OREGAMI canned mapping using regular structure of TCG

Figure 7: Gantt charts of divide and conquer binomial tree mapped to 4 processor hypercube

and scheduling, debugging, and performance analysis. One of the challenges facing researchers in parallel processing today is the problem of integrating software tools that have been developed independently within each of these domains.

Below we give several examples of how the TCG could be used within parallel programming environments. We note that a number of existing projects utilize an abstraction that is compatible with that of the TCG. These include languages such as OCCAM, Data Parallel C, C and Fortran with communication extensions, Dino and Z; parallel programming environments such as Prep-P [4], ORCA [10], the Parallel Programming Environments Project [2], and TIPS [29]; and commercial systems such as the Intel hypercube machines, NCUBE hypercube, Intel iWarp, and Transputer.

Debugging

The TCG has the potential to aid in the debugging of parallel programs when used as a specification of the intended operation of the code. In other words, the logical view of events presented by the TCG and the phase expression can be used to analyze the execution traces which report the actual sequences of events at run time. For example, in Figure 8 comparison of the LaRCS TCG to the runtime TCG for the *7-body algorithm* reveals an errant message and a missing iteration of the inner loop. Specifically, process 2 erroneously sent to process 4 (instead of process 3) in the ring phase. Also, there are only two iterations of the inner loop when there should be three.

This approach is utilized in Hough and Cuny's work with the Belvedere pattern-oriented debugger [11] [12]. In their system, user-defined abstract events are viewed through one or more 'perspectives,' thereby facilitating the debugging of the parallel program through visualization and pattern matching techniques. We note that the LaRCS comphase corresponds precisely to one of the abstract events monitored by their system (logical communication events). Furthermore, we note that the three ordering relations they define (*precedes, parallel,* and *overlaps*) are very similar (but not identical) to the three operators used in the LaRCS phase expressions (*precedes, parallel,* and *span*) [7]. We believe that the TCG can be effectively utilized in parallel debuggers with foundations in Lamport's model such as Belvedere and [9].

[7] We developed our operators independently although the similarity most certainly arises from the common foundations in Lamport's process time graphs and Allen's temporal relations.

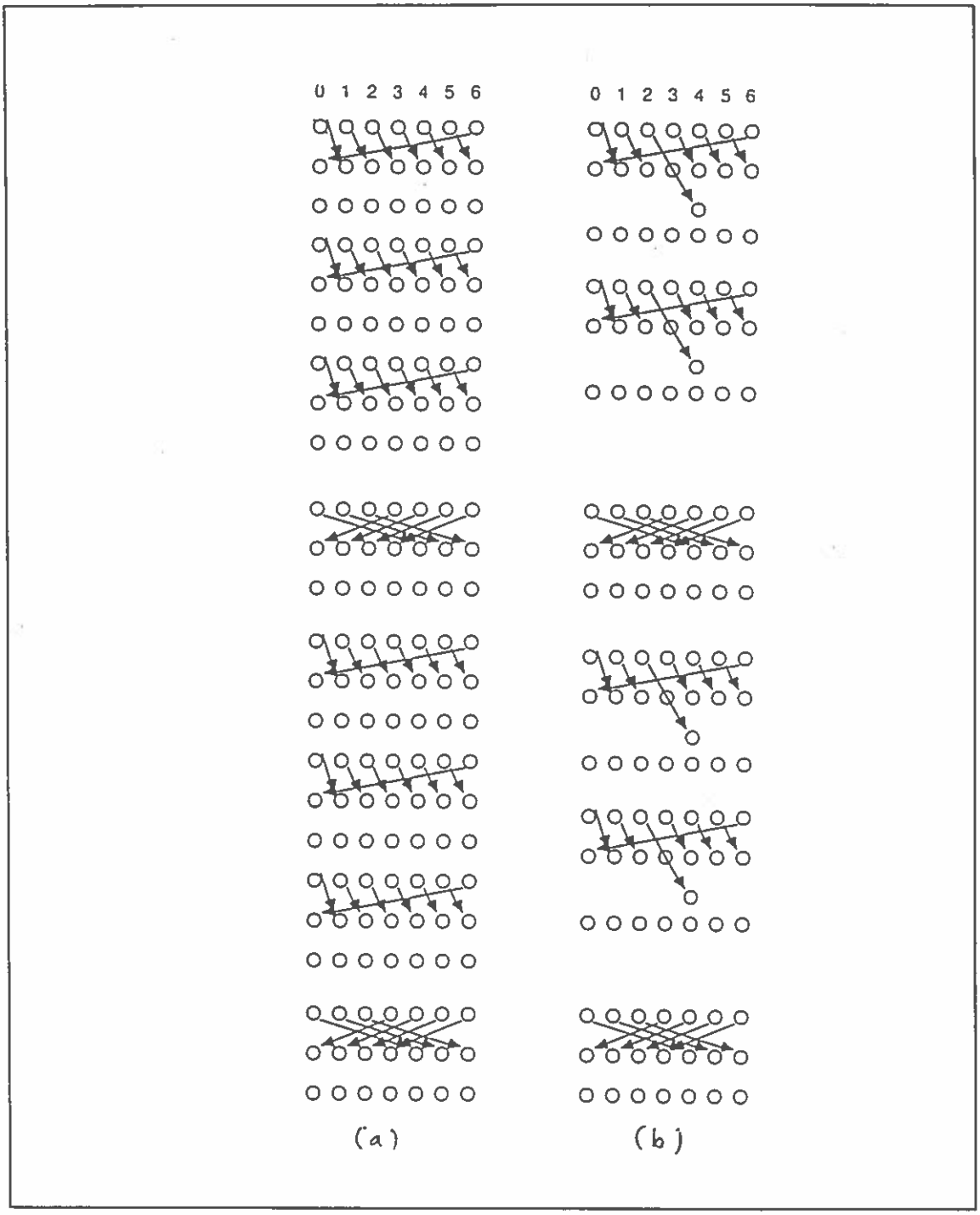


Figure 8: (a) TCG for the *n-body algorithm* (b) runtime trace for the *n-body algorithm* Note: intraprocess edges have been omitted for clarity.

Performance Analysis

The TCG can be effectively utilized for static and dynamic analysis of the performance of parallel computations. For example, in the OREGAMI system, the phase-expression is used to drive the evaluation of the mapping of a given parallel computations using the well-known metric of *completion time.* OREGAMI estimates the completion time of a mapped computation on a phase by phase basis using utility functions that traverse the abstract syntax tree representation of the phase expression. This calculation takes into account time-related characteristics which include characteristics of the computation defined in the LaRCS code, characteristics of the target architecture such as channel bandwidth, message-startup time, message-switching time, and routing scheme (such as store-and-forward, virtual cut-through, or wormhole), and the mapping itself.

The LaRCS phase expression also has the potential to drive a dynamic animation of the computation, with appropriate visual cues to represent the active compute and communication phases, as well as performance bottlenecks such as hot spots, load imbalance, etc.

TCG as an abstraction for parallel programming

We note that the TCG model of parallel computation fits the XYZ abstraction proposed by [25] and that it is possible that the LaRCS code definition of the TCG can be embedded in languages that fit the XYZ abstraction, such as the proposed language *Z* and its *ensembles* abstraction [10]. The ensembles `section` indexing scheme is analogous to the `nodetype` labeling scheme of LaRCS; the port ensembles declarations and communication graph bindings are similar to the `comtype` declarations of LaRCS. While the XYZ notion of *phases* as the "parallel composition of a set of processes" differs from the TCG notion of *phases* (logically synchronous communication or computation), it is possible to view the LaRCS phase expression as Z level code. Because of the undesirability of requiring users to write the LaRCS code separately from the actual program code, it seems advisable to be able to integrate the two by adding LaRCS language constructs to the host language.

6 Summary

Expressive power. The TCG was designed for medium to large grained parallelism, for a process-oriented/message-passing view of parallel computation, and is especially suitable for regular parallel computations that can be described in terms of phases of logically synchronous computation and com-

munication. However, the TCG is also capable of modeling computations of arbitrary granularity. The TCG offers the unique capability of identifying compute and communication phases, and of describing the temporal behavior of the parallel algorithm in terms of these phases. The limitations of the TCG include its failure to model conditional, non-determinism, and the necessity to serialize multicast communication.

Utility. Because the TCG allows mapping and scheduling algorithms based on the two predominant models to cross domains, it provides greater flexibility in the use of mapping software. In addition, because of its integration with the Lamport process-time graph, we believe the TCG provides an abstraction that will make it useful in general parallel programming environments: for program development, debugging, and performance monitoring.

Efficiency. The LaRCS language provides an efficient representation of the TCG for regular parallel computations through its ability to represent parameterized families of graphs. Thus regular computations can be represented by LaRCS code whose size is independent of the size of the TCG. The phase expression provides an efficient representation of the TCG phase-DAG which can be used to generate successive portions of the TCG as needed. The current LaRCS translator does not generate a full instantiation of the TCG, but instead generates C functions that can be invoked by OREGAMI utility functions to generate a portion of the TCG or the full TCG-DAG.

Performance. We are currently conducting experiments to compare the performance of static task graph algorithms and DAG-based algorithms with that of OREGAMI's algorithms. We note that the TCG makes these comparative experiments possible. We are continuing to develop new mapping and scheduling algorithms in order to achieve significant gains in the mapping of parallel algorithms to parallel architectures.

7 Acknowledgments.

Thanks to Xiaoxiong Zhong, Jan Telle, Sanjay Rajopadhye and other members of the OREGAMI research group, Fran Berman, and Larry Snyder for discussions that contributed to this paper.

References

[1] G. Agha. *Actors.* Springer-Verlag, 1986.

[2] 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.

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

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

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

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

[7] E. G. Coffman. *Computer and Job-Shop Scheduling Theory.* Wiley Interscience, 1976.

[8] 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.

[9] C. J. Fidge. Partial orders for parallel debugging. In *SIGPLAN Notices*, pages 183–194, January 1989.

[10] W. G. Griswold, G. A. Harrison, D. Notkin, and L. Snyder. Port ensembles: a communication abstraction for nonshared memory parallel programming. Technical report, University of Washington, 1989.

[11] A. A. Hough and J. E. Cuny. Initial experiences with a pattern-oriented parallel debugger. In *SIGPLAN Notices*, pages 195–205, January 1989.

[12] A. A. Hough and J. E. Cuny. Perspective views: A technique for enhancing parallel program visualization. In *Proceedings IEEE 1990 International Conference on Parallel Processing*, pages II:124–132, August 1990.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] C.Y. Lee, J. J. Hwang, Y. C. Chow, and F. D. Anger. Multiprocessor scheduling with interprocessor communication delays. *Oper. Res. Letters*, 7(3):141–147, 1988.

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

[16] 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.

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

[18] 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.

[19] V. M. Lo, S. Rajopadhye, M. A. Mohamed, S. Gupta, B. Nitzberg, J. A. Telle, and X. X. Zhong. LaRCS: A language for describing parallel computations for the purpose of mapping. Technical Report CIS-TR-90-16, University of Oregon Dept. of Computer Science, 1990.

[20] P. A. Nelson and L. Snyder. Programming paradigms for nonshared memory parallel computers. In *The Characteristics of Parallel Algorithms*, pages 3–20. The MIT Press, 1987.

[21] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.*, 19(2):322–328, April 1990.

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

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

[24] V. Sakar. Partitioning and scheduling parallel programs for execution on multiprocessors. Technical report, Ph.d. Thesis, Dept. of Computer Science, Stanford University, 1987.

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

[26] L. Snyder. The xyz abstraction levels of poker-like languages. In *Languages and Compilers for Parallel Computing*, pages 470–489, August 1989.

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

[28] J. A. Telle, V. Lo, and S. Rajopadhye. Applications of group theory to the mapping problem in distributed memory machines. in preparation.

[29] B. Veltman, B.J. Lageweg, and J.K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.

[30] A. Wagner, S. Chanson, N. Goldstein, J. Jiang, H. Larsen, and H. Sreekantaswamy. Tips: Transputer-based interactive parallelizing system. Technical report, Dept. of Computer Science, University of British Columbia, 1990.

[31] X. Zhong and V. Lo. Application dependent routing algorithms for distributed memory machines. submitted to PARLE-92.