Impact of Message-Ordering in Wormhole-Routed Multicomputers

Dhabaleswar K. Panda and Vibha Dixit-Radiya

Technical Report
OSU-CISRC-9/94-TR20

A preliminary version of this paper was presented at *Scalable High Performance Computing Conference (SHPCC '94).*

Manuscript is under review for *Journal of Parallel and Distributed Computing.*

Impact of Message-Ordering in Wormhole-Routed Multicomputers *

Dhabaleswar K. Panda and Vibha Dixit-Radiya

Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210
E-mail: {panda,radiya-v}@cis.ohio-state.edu

Abstract

This paper analyzes the impact of message-ordering, between outgoing messages from a sender to multiple receivers (called multicasts), on the completion time of a program for distributed-memory systems. We study how best to order a set of outgoing messages by taking into account message criticality and architectural issues like multiple injection ports, routing adaptivity, and link contention. First, the simple message-ordering algorithm of Dikaiakos et al. [8], proposed only for fully-connected one-port system, is enhanced to obtain a *static* message-ordering algorithm. This enhanced algorithm is based on message criticality obtained from the temporal dependencies of the program and works for non-fully-connected systems. Next, a *dynamic* message-ordering algorithm is proposed which additionally considers architectural issues including link contention and multiple ports for wormhole-routed systems. Both algorithms are compared against naive sequential message-ordering on an event-driven simulator for Gaussian Elimination and random task graphs. Simulation experiments show a reduction in program completion time up to 33.1% by using static over sequential ordering. Similarly, the dynamic ordering shows additional improvement of up to 17.7% over static ordering. This research indicates that good message-ordering schemes need to be used as the last step in the mapping and scheduling process to generate efficient programs for wormhole-routed distributed-memory systems.

Index Terms — message-ordering, wormhole-routing, multicast, link contention, multiport, adaptive routing, temporal communication graph, mapping.

*A preliminary version of this work was presented at Scalable High Performance Computing Conference (SHPCC'94). This research is supported in part by the National Science Foundation Grant # MIP-9309627.

Contents

1 Introduction

One of the major problems in software development for parallel systems is the mapping problem [3, 7, 10, 16, 19, 25]. It is defined as allocation of the set of tasks of a parallel application onto available processors to obtain minimal program completion time. The mapping problem being NP-hard, is usually solved by solving its subproblems, viz., (a) *clustering* — grouping of tasks for bounded number of processors, (b) *assignment* — one-to-one mapping of task clusters onto processors with specific topology, (c) *intra-processor scheduling* — ordering of ready computations of tasks allocated to the same processor, and (d) *message-ordering* — ordering between outgoing messages from a sender to multiple receiver processors. There has been extensive work in the literature on developing near-optimal heuristics for subproblems (a) and (b) above. Even though subproblems (c) and (d) also significantly affect program completion time, they have not received any attention in mapping literature. For distributed-memory systems, we have solved the clustering and intra-processor scheduling problems in [10] and the assignment problem in [9]. In this paper, we solve the message-ordering problem for task graphs for which clustering, assignment, and scheduling solutions have already been obtained.

Logical communication between tasks in a parallel program can be categorized as either *unicast* (source sending a message to a destination) or *multicast* (source sending messages to multiple destinations at the same time). A *broadcast* is a special case of multicast. Multicasts are further classified into *personalized* (different data going to others) and *non-personalized* (same data going to different processors). Personalized multicasts are common in scientific and numerical computations using *scattering* techniques. All-to-all personalized communication operation is used in parallel fast Fourier transform, matrix transpose, and parallel database join operations [15]. Sophisticated multicasting schemes using path-based routing [21, 22] and unicast-based schemes [14] have been shown to be efficient for non-personalized multicast. However, personalized multicasts cannot take advantage of these schemes. Hence for systems not having support for these schemes and for personalized multicasts, there is no other alternative but to send messages from the source node to destination processors as a collection of unicast messages. In such cases, the order in which these messages are sent from the source has direct impact on the program completion time. Figure 1 shows parts of an example distributed-memory parallel program with 4 tasks. Consider the second multicast with messages m_1 and m_2 originating from source node s and reaching destination nodes d_1 and d_2. Suppose the length of path p_2 is greater than that of path p_1. Then, in the second multicast, sending m_2 *before* m_1 is a better ordering than m_1 *before* m_2, if program completion time is to be minimized.

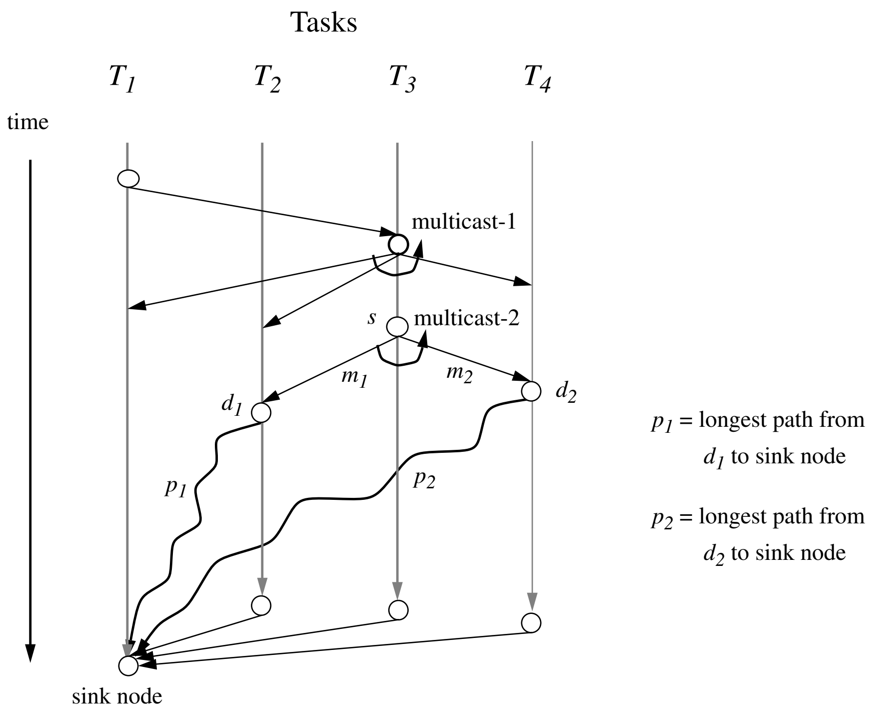

Figure 1: Ordering of messages in a multicast in a distributed-memory parallel program affects overall program completion time.

The wormhole-routing switching technique is increasingly becoming popular in building massively parallel systems due to its inherent advantages like low-latency communication and reduced communication hardware overhead [21]. In addition to the basic wormhole-routing switching, systems are gradually incorporating multiple communication ports. Intel Paragon [18], Cray T3D [5], and Stanford DASH [13] are some early representative systems in this trend. These systems provide low-latency communication when the traffic in the system is low. However, with increase in communication traffic, messages undergo severe *link contention* and the system starts performing poorly. Similarly, when a single processor sends multicast messages, they encounter *port contention* at the router of the sender node depending on the routing strategy used by the system. Such contention may increase the completion time of a program execution significantly. Such increase in completion time can be reduced by determining an effective *message-ordering* strategy. Though commercial wormhole systems are becoming available, there is no study in the literature about the *interplay* between link contention and multiple ports and their impact on determining a suitable message-ordering strategy. In this paper, we take such an approach in determining effective message-ordering strategy for wormhole-routed systems. For future machines, we also study the effect of adaptivity in routing on message-ordering. The problem of message ordering is similar to that of instruction reordering in pipelined processors to minimize sequential program execution time [17].

The only known prior work on message ordering attempting to reduce program completion time is by Dikaiakos et al. [8]. They present an optimal message-ordering algorithm that achieves minimal program completion time for a given clustered, mapped, and scheduled task graph. The algorithm is based on criticality of messages. However, the algorithm assumes a fully-connected architecture and a single port communication model. This makes the results inapplicable to realistic architectures which are not fully-connected.

In this paper, we first enhance the strategy proposed by Dikaiakos et al. to non-fully-connected architecture and evaluate its performance compared to the naive *sequential* message-ordering strategy. Then we propose a new *dynamic* message-ordering algorithm based on link contention and port availability. We analyze the performance of these algorithms under two different models of communication start-up: *skewed* and *concurrent*. We study the effectiveness of these algorithms against sequential message-ordering scheme for random task graphs with varying computation to communication characteristics and for systems with multiple ports and varying routing adaptivity. Overall, message-ordering schemes for wormhole-routed systems are proposed by taking into account (a) criticality of messages in the application, (b) link contention, and (c) number of ports in the processor-router interface. Ordering is obtained within multicast(s) (a parallel program can have many multicasts) and the goal of the problem is to minimize overall program completion time.

The paper is organized as follows. Section 2 discusses the significance of message ordering in wormhole systems under link and port contention. Section 3 covers preliminaries on Temporal Communication Graph model and communication startup cost models. Message ordering algorithms are presented in section 4. Simulation experiments and results are presented in section 5. The conclusions and future work are presented in section 6.

2 Message-Ordering in Wormhole Routed Systems

In this section, we introduce the basic concept of wormhole routing, its sensitivity to link contention, and how adaptivity in routing reduces link contention. We discuss the operational principles of multiport wormhole systems and show situations which lead to port contention. Through an example, we show the significance of message-ordering by taking into account link contention, multiple ports and routing adaptivity.

2.1 Link Contention and Routing Adaptivity

In wormhole-routed systems [6], the header flit of a message establishes the path, the intermediate flits follow the path, and the tail flit releases the path. During the message propagation, if a desired link is already being used by another message, the current message gets blocked. This message waits in the network occupying all the links it is traversing. Such a phenomena is known as *link contention.* Link contention is closely related to the underlying *routing* scheme, topology of the system, and the communication traffic.

To alleviate link contention, several routing schemes with varying *adaptivity* have been proposed in the literature. *Deterministic* or *e-cube* routing [6] defines a single path from a source to a destination node and thus has zero adaptivity. Such routing is simple to implement and deadlock free. However, it does not make effective use of all communication links in a system. *Fully adaptive* algorithms [11] allow a message to be routed along any of the shortest paths from the source to the destination processor. Partially adaptive algorithms like *planar* [4] restrict routing freedom to two dimensions at a time.

Figure 2 illustrates the differences between these three routing schemes for a 3-dimensional cube. Higher adaptivity has potential to reduce link contention and hence is useful to reduce the overall execution time of a given program. However, for any of the above schemes, the system performance also depends on how messages are injected into the network or taken out from the network by the *processor-router interface.* The number of available ports at this interface plays a significant role in determining the system performance and hence the completion time of a program. With limited number of ports at a processor-router interface, there is added chance that a message will undergo *port contention* in addition to link contention. Hence, a good message ordering strategy should consider link contention, port contention, and routing adaptivity together. Though current generation architectures use only deterministic or e-cube routing, adaptive schemes will be used in near future machines. Hence, we emphasize on both deterministic and adaptive routing schemes in this paper.

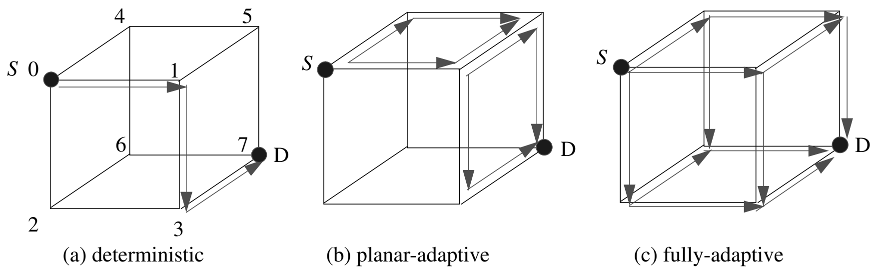

Figure 2: Possible paths from a source to a destination under different wormhole routing schemes in a 3-cube.

2.2 Contention in Multiport Systems

Consider a typical processor-router organization in a multicomputer with two *injection* and two *consumption* ports (channels) as shown in Fig. 3. Any message originating at a processor must be sent via an injection port to the router [21]. This port remains busy till the message leaves the router. If the message gets blocked on its way due to link contention, the assigned port remains busy and cannot be allocated to any other message. This leads to messages being queued at the injection port(s) introducing port contention. With multiple ports, the contention may get alleviated. However, it does not get eliminated completely because the number of outgoing messages at any time may be greater than the number of available injection ports. Similar contention due to limited number of consumption ports also degrade system performance significantly. This aspect has been studied separately in [1]. In this paper we focus on contention due to injection ports and determine best ways to utilize injection ports in a system together with its routing scheme.

2.3 Effect of Message-Ordering

Consider an example multicast pattern of Fig. 4 in a 4x4 mesh. The source processor $P6$ sends 6 messages, m_0, m_1, m_3, m_{12}, m_{14}, and m_{15} to 6 respective destinations. Assume *ecube* routing [6] is

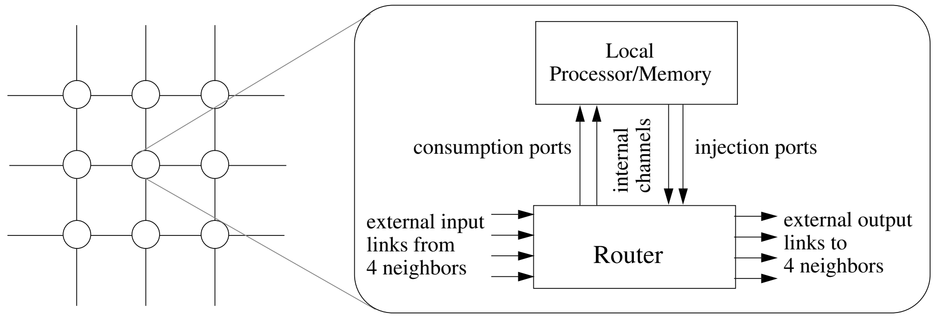

Figure 3: Processor and router organization of a node supporting two-port communication.

being used for determining routing paths. It can be seen that multiple outgoing messages may contend for the same outgoing link (messages m_0, m_1, m_{12} for the westbound link from $P6$ and messages m_3 and m_{15} for the eastbound link). Since messages come to the router through injection ports, the order in which they get propagated from the router to the network is based on (a) the order in which the processor prepares the messages to go out and (b) the number of ports available in the system.

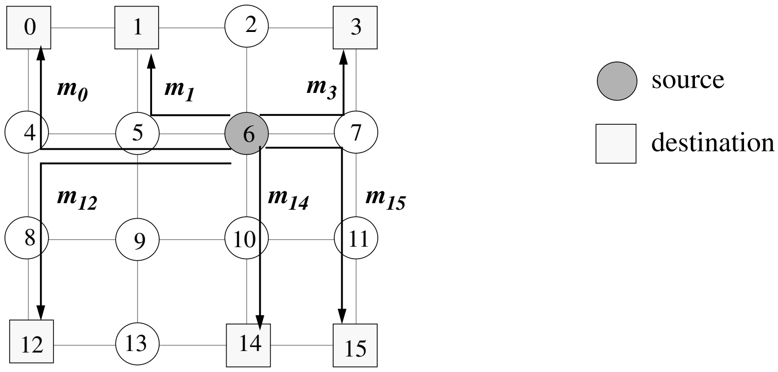

Figure 4: An example ordering of a multicast pattern in a 4x4 mesh with ecube routing.

Assume that in Fig. 4, the messages in the multicast pattern are the only messages in the network. If the system supports only 1 injection port, then the order of message propagation is identical to the *message-order* prepared by the processor. For example, a message-ordering of m_0, m_1, m_3, m_{12}, m_{14}, and m_{15} will force the messages to get propagated in that order. If the system supports 2 ports it can be observed that the above message-ordering is not efficient. Both messages m_0 and m_1 will grab the two injection ports. Due to link contention, m_0 will propagate and m_1 will get blocked. This is a poor utilization of ports. It can be noticed that a message-ordering of $(m_0, m_3, \ldots)$ will allow both m_0 and m_3 to propagate simultaneously by using the two ports. If the underlying routing scheme is *fully-adaptive*

[11], then the original message-ordering $(m_0, m_1, \ldots)$ would have allowed both messages m_0 and m_1 to move simultaneously. Hence, there is a close interplay between the routing scheme, number of injection ports, link contention, and the latency of messages sent in a multicast.

Even for an 1-port system, suitable message-ordering can increase the port utilization by assigning the port to a message whose outgoing link is free. Besides utilization of ports, from an application perspective, there exists *criticality* in messages, *i.e.*, some messages are more critical than others and the program completion time increases significantly [8, 9] if the critical messages get delayed. Hence, a good message-ordering algorithm should take into account factors such as criticality in messages, link contention, port contention, and routing scheme, to reduce program completion time as well as increasing the utilization of ports. Figure 5 shows the cause-effect relationship between the various factors.

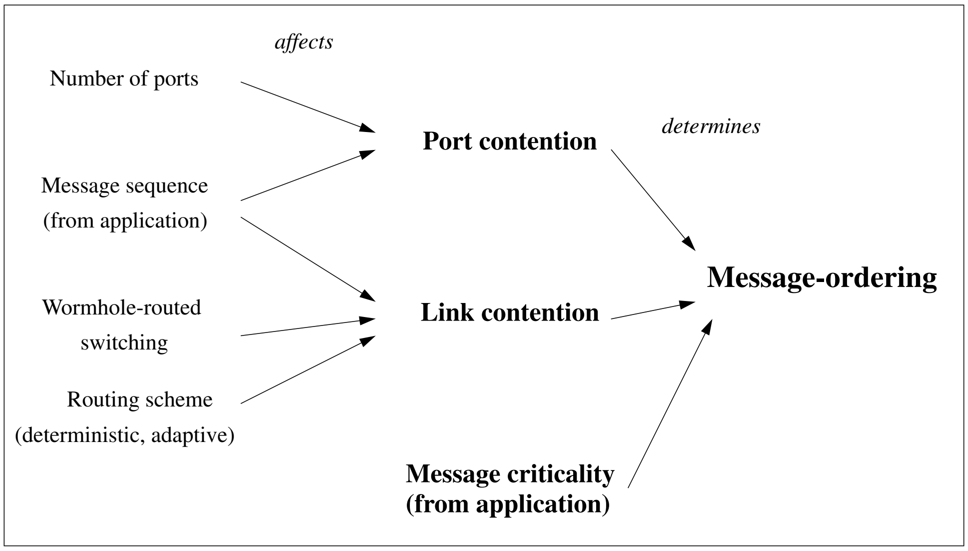

Figure 5: Relationship between factors determining message ordering.

3 Preliminaries

In deriving message ordering algorithms, we take an application perspective and emphasize on program completion time. In a typical program, multiple multicast communication steps happen at different points in the program execution. Since there are direct and indirect temporal dependencies through communication steps in the program execution, our objective here is to reduce the overall program execution time by deriving an effective message-ordering strategy. We assume that clustering, intra-processor scheduling, and assignment solutions for the parallel program have already been obtained.

3.1 Temporal Communication Graph (TCG)

We solve the message-ordering problem for distributed-memory systems for which programs are *explicitly-parallel* in nature [10]. The temporal dependencies in an explicitly-parallel program are modeled very well by a *Temporal Communication Graph* (TCG). Figure 6 shows an example explicitly-parallel program with 4 tasks and Fig. 7a shows its TCG model. A TCG consists of a set of tasks where each task is a sequence of computation nodes. Node weights represent amount of computation performed in the corresponding computation. Intra-task edges (with zero weights) denote sequencing within a task. Inter-task edges denote explicit communication between tasks and weights on these edges represent size of associated message transfers. The earliest and latest time estimates shown in the figure will be in the next section.

Task T_1	Task T_2	Task T_3	Task T_4

{
compute [300];
send (T_2, 10);
send (T_3, 10);
recv (T_2, 10);
recv (T_3, 10);
compute [100];
send (T_4, 10);
}
	{
	recv (T_1, 10);
	compute [50];
	send (T_3, 30);
	compute [50];
	send (T_1, 10);
	}
		{
		recv (T_1, 10);
		compute [100];
		recv (T_2, 30);
		compute [20];
		send (T_1, 10);
		}
			{
			compute [10];
			recv (T_1, 10);
			compute [170];
			}

Figure 6: An example explicitly-parallel program. The associated numbers for compute and send/recv statements indicate the number of consecutive instructions executed and size of message transferred, respectively.

The TCG model is an improvement over the two existing models – nontemporal *interaction graph* model [2, 3] used for explicitly-parallel programs, and temporal *precedence graph* model [16, 12, 20, 26] used for sequential programs. The TCG model has been successfully used in our previous study of clustering and assignment problems [9, 10]. Details of the model can be found in these references. We solve the message-ordering problem assuming that clustering, assignment, and intra-processor scheduling solutions have already been obtained for the TCG. Hence, it is assumed that the number of tasks in a TCG equals the number of available processors in the host multicomputer. Figure 7b shows the

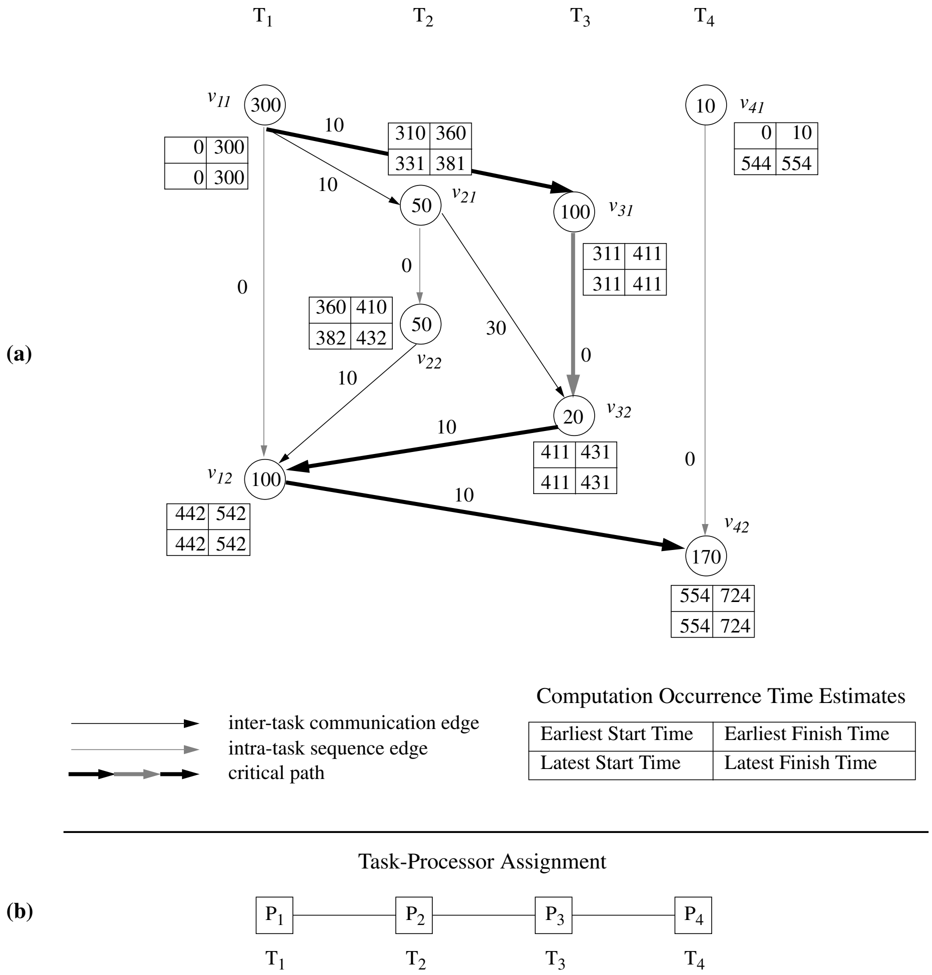

Figure 7: Temporal communication graph in (a) with earliest and latest computation times for the example program. The estimates are based on the task-processor assignment shown in (b).

assignment of tasks to processors of a linear array. Note that for explicitly-parallel programs, it is the tasks and not computation nodes that are mapped onto processors to prevent data migration.

3.2 Communication startup cost models

Depending on the design of the processor-router interface and software components associated with the communication primitives in a system, there can be two different cost (delay) in the way messages are injected into the network. We identify them as *concurrent start-up* and *skewed start-up* cost models. In the first model, all messages belonging to the multicast pattern from a single node are prepared and presented to the processor-router interface concurrently with a single communication start-up. In the second model, the messages are prepared and presented one after another by encountering start-up for each message. Thus, the messages are presented to the processor-router interface in a skewed manner. Figure 8 shows the message injection delays for the two communication cost models. We consider both these models in our analysis and comparison of various message-ordering strategies.

The injection ports in a system are nothing but DMA channels. For concurrent startup model, the messages of a multicast are released into the communication network simultaneously, though some of them can be blocked on their first link due to link contention. It is important to note that the concurrent startup cost model is possible only if multiple DMA channels can simultaneously make copies (for different destinations) of a single message (non-personalized multicast) or multiple messages (for personalized multicast) from the user buffer to the system buffer in memory. Current generation systems do not yet support such simultaneous operations without performance degradation. Thus, the skewed startup cost model is a more realistic model for current generation multicomputers. Hence, we focus more on the skewed model in our analysis.

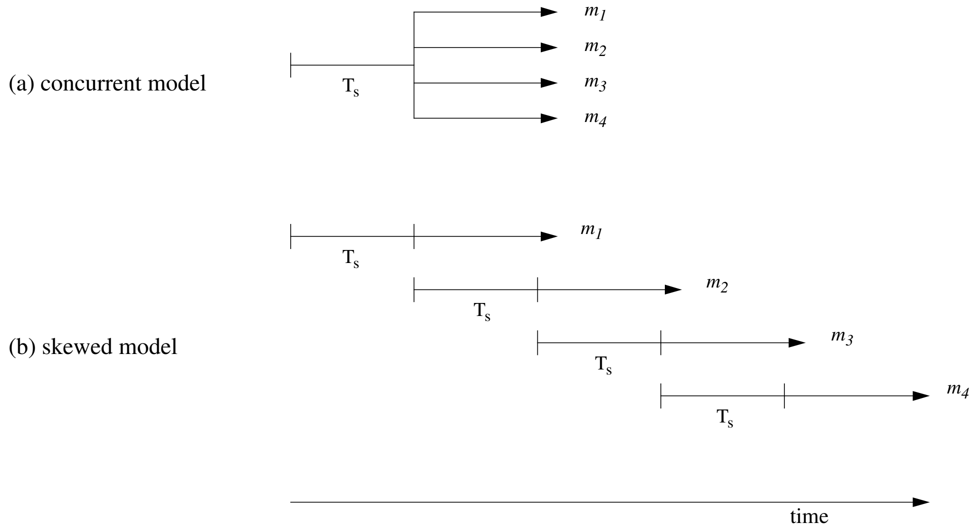

Figure 8: Communication startup cost models: (a) concurrent model with simultaneous message injection into network and (b) skewed model with staggered message injection.

4 Message-Ordering Algorithms

In this section, we present our message-ordering schemes. First we explain the naive *sequential* ordering strategy. Next, we discuss the work done by Dikaiakos et al. [8] which presents a message-ordering scheme based on precedence graph model for 1-port fully-connected system (not a realistic architecture

Table 1: Summary of Notations

Symbol	**Meaning**
TCG	Temporal Communication Graph.
T_x	Task x in TCG.
V	Set of nodes in TCG.
E	Set of edges in TCG.
E_c	Set of communication edges in TCG.
v_{xp}	a node in V representing pth computation stage in T_x.
$e_{xp,x(p+1)}$	a sequence edge in E from v_{xp} to $v_{x,p+1}$.
$e_{xp,yq}$	a communication edge in E_c from v_{xp} to v_{yq} $(x \neq y)$.
$wcomp(v_{xp})$	amount of computation in node v_{xp}.
t_x	time to execute one unit of computation by a processor.
$tcomp(v_{xp})$	time to compute $v_{xp} = wcomp(v_{xp}) \times t_x$
$wcomm(e_{xp,yq})$	amount of communication in edge $e_{xp,yq}$.
t_s	message startup cost.
t_{phy}	physical link propagation delay.
t_{sw}	switching delay through a node = delay in transferring data through a node.
t_{node}	node delay through a router = time to make routing decision.
$WH_{ideal}(len, dist)$	Wormhole routing ideal latency for a message of size len and sent over $dist$ hops $= t_s + (t_{node} + t_{phy}) * dist + (t_{sw} + t_{phy}) * (len - 1)$.
$tcomm(e_{xp,yq})$	ideal latency of message in $e_{xp,yq}$.
EST, EFT	Earliest Start Time, Earliest Finish Time.
LST, LFT	Latest Start Time, Latest Finish Time.
CT	Completion Time of a TCG.
CCR	Computation-to-Communication Ratio of a TCG.

for large-scale systems). Using the TCG model, we extend their result to obtain a *static* algorithm for wormhole-routed distributed-memory systems with any topology. We then develop a new *dynamic* algorithm to obtain better message ordering by taking into account architectural characteristics of a system like routing scheme, link contention, and port contention. Simulation experiments and results showing the impact of our message-ordering algorithms on various applications are presented in the next section. Table 1 shows the summary of notations used in this paper.

4.1 Sequential Ordering

This is a naive message-ordering scheme which is used in the absence of knowledge about application or architectural characteristics. Under this ordering, messages in a multicast are sent by the router in an increasing order of receiving task identifiers. For example, for the TCG in Fig. 7a, the messages in the multicast originating from source v_{11} is sent in the order v_{21} and v_{31} (corresponding to tasks T_1 and T_2, respectively). For the multicast pattern in the 4x4 mesh in Fig. 4, the ordering would be m_0, m_1, m_3, m_{12}, m_{14}, and m_{15}. We compare our static and dynamic schemes with this sequential ordering scheme to illustrate performance improvement.

4.2 Static Ordering

The static message-ordering is based on criticality of messages from the perspective of program completion time. A quantitative measure of criticality is estimated by *Latest Start Time* (LST). Given a TCG, a multicomputer (not necessarily fully-connected), and an assignment of tasks, LST values are obtained from the *occurrence time estimates* of computation nodes in a TCG. These include Earliest Start Time (EST), Earliest Finish Time (EFT), Latest Start Time (LST), and Latest Finish Time (LFT) which are formally defined below. Here, v_{xp} denotes the pth computation node in task T_x, $e_{xp,yq}$ denotes a communication edge from v_{xp} to v_{yq} ($x \neq y$), $tcomp(v_{xp})$ denotes the time to compute v_{xp}, and $tcomm(e_{xp,yq})$ denotes the ideal latency (without link contention) for message transfer for edge $e_{xp,yq}$. For wormhole-routed switching, the latency for a message of size *len* flits communicated between processors that are distance *dist* apart, is given by $WH_{ideal}(len, dist) = (t_s + (t_{node} + t_{phy}) * dist + (t_{sw} + t_{phy}) * (len - 1))$, where t_s is the message startup cost, t_{node} is the node delay through a router, t_{phy} is the physical link propagation delay, and t_{sw} is the delay in transferring data through a node.

$$EST(v_{xp}) = \begin{cases} 0, & \text{if } v_{xp} \text{ is a source node;} \\ max_{\forall y,q}[EFT(v_{yq}) + tcomm(e_{yq,xp})], & \text{if } v_{xp} \text{ is not a source node \&} \\ & \quad v_{yq} \text{ is a predecessor of } v_{xp}. \end{cases}$$

$$EFT(v_{xp}) = EST(v_{xp}) + tcomp(v_{xp}).$$

$$earliest\ completion\ time = max_{\forall x,p}[EFT(v_{xp})], \qquad \text{if } v_{xp} \text{ is a sink node.}$$

$$LFT(v_{xp}) = \begin{cases} earliest\ completion\ time, & \text{if } v_{xp} \text{ is a sink node;} \\ min_{\forall y,q}[LST(v_{yq}) - tcomm(e_{xp,yq})], & \text{if } v_{xp} \text{ is not a sink node \&} \\ & \quad v_{yq} \text{ is a successor of } v_{xp}. \end{cases}$$

$$LST(v_{xp}) = LFT(v_{xp}) - tcomp(v_{xp}).$$

Informally, the earliest times are generated by scheduling each computation and communication at the earliest possible time. The latest time estimates are then generated by scheduling computation and communications at the latest possible time such that the earliest completion time does not increase. In Fig. 7a, the earliest and latest time estimates of computation nodes are shown assuming that the computation time of nodes is equal to the corresponding node weights and communication time cost (in the same units) is equal to $WH_{ideal}(len, dist) = len + dist - 1$ (i.e. $t_{phy} = 1$ and $t_s = t_{node} = t_{sw} = 0$). Details of the occurrence time estimates can be found in our study of the mapping problem in [9, 10].

For the *static LST-based message ordering*, we order messages within a multicast in increasing order of the LST estimates of the message's destination nodes. This is because lower the LST value, the higher is the criticality of the message with respect to completion time and our goal is to reduce program completion time. For example, for the TCG in Fig. 7, the messages in the multicast originating from source v_{11} is sent in the order v_{31} and v_{21} (LST(v_{31}) = 311 < LST(v_{21}) = 331) under the static message-ordering scheme. It is to be noted that this message ordering is different from the sequential message ordering.

Observation 1 *The time complexity of the static message ordering algorithm is given by the complexity of computing LST (Latest Start Time) estimates of a TCG (Temporal Communication Graph). This is equal to $\mathcal{O}(|V| + |E|)$, the complexity of topological sort, where $|V|$ and $|E|$ are the number of nodes and edges, respectively, in the TCG.*

In the algorithm by Dikaiakos et al. [8], messages are ordered in decreasing order of *Finish Times* of the message's destination nodes. It turns out that Finish time of a node is same as the earliest completion time minus the node's Latest Start Time, when the topology is fully connected. Dikaiakos's algorithm can thus be applied only for fully-connected architectures while our static algorithm works for non-fully connected architectures as well. This is reflected in the calculation of $tcomm$ values for communication edges. In our preliminary simulation experiments, it was found that the performance (in terms of program completion time) of Dikaiakos' algorithm was a little worse than or at best equal to that of the static algorithm. Hence, we have focused on static algorithm, instead of Dikaiakos' algorithm, in the simulations. Another point to note is that Dikaiakos et al. have solved the message ordering problem for sequential programs represented by precedence graph model (also called directed acyclic graph) while we address the problem for explicitly-parallel programs represented by TCG model. In the next section, we present a more interesting dynamic message-ordering algorithm which is a further improvement from the static algorithm.

5 Dynamic Ordering

In the static message ordering scheme, message criticality is determined by the LST estimates. These are computed assuming ideal message latencies for communication cost of edges in a TCG. Thus, the static scheme ignores the effect of link contention on message criticality. In general, a parallel program would face significant amount of link contention between messages belonging to same or different multicasts. The dynamic message algorithm presented here attempts to further reduce program completion time by considering actual message latencies. This is done by examining link contention and port contention when deciding the order in which messages must be scheduled.

The basic concept behind the *dynamic* ordering is not to assign a port to a message which cannot propagate out from the router due to link contention. This is achieved by incorporating a *message-scheduler* with each processor-router pair as shown in Fig. 9. The scheduler maintains a queue of unicast and LST-ordered multicast messages which it receives continuously from the application task running on the processor. Thus, messages are ordered in decreasing order of their criticality with respect to the application. Under the concurrent startup communication cost model, as discussed in section 3.2, a multicast encounters a single startup cost for all its messages before they are submitted to its processor's message-scheduler. Under the skewed startup cost model, each message in a multicast encounters a startup cost resulting in skewed submissions of messages (within a multicast) to its message-scheduler.

The message scheduler on each processor functions as follows. Whenever an injection port on its node becomes free, it selects the earliest (first) message from the queue having the least LST. The scheduler queries the node's router for the identity of the external output link on which this message will be sent. The router decides the output link based on the embedded routing algorithm, deterministic or adaptive, and the current network status (in case of adaptive routing). If the selected output link is free, then the earliest message is scheduled by assigning it the free port and submitting the message to the router for actual transmission. If the output link for the earliest message is not free, the message scheduler holds back this message (in the queue) and examines subsequent messages in the queue one after another.

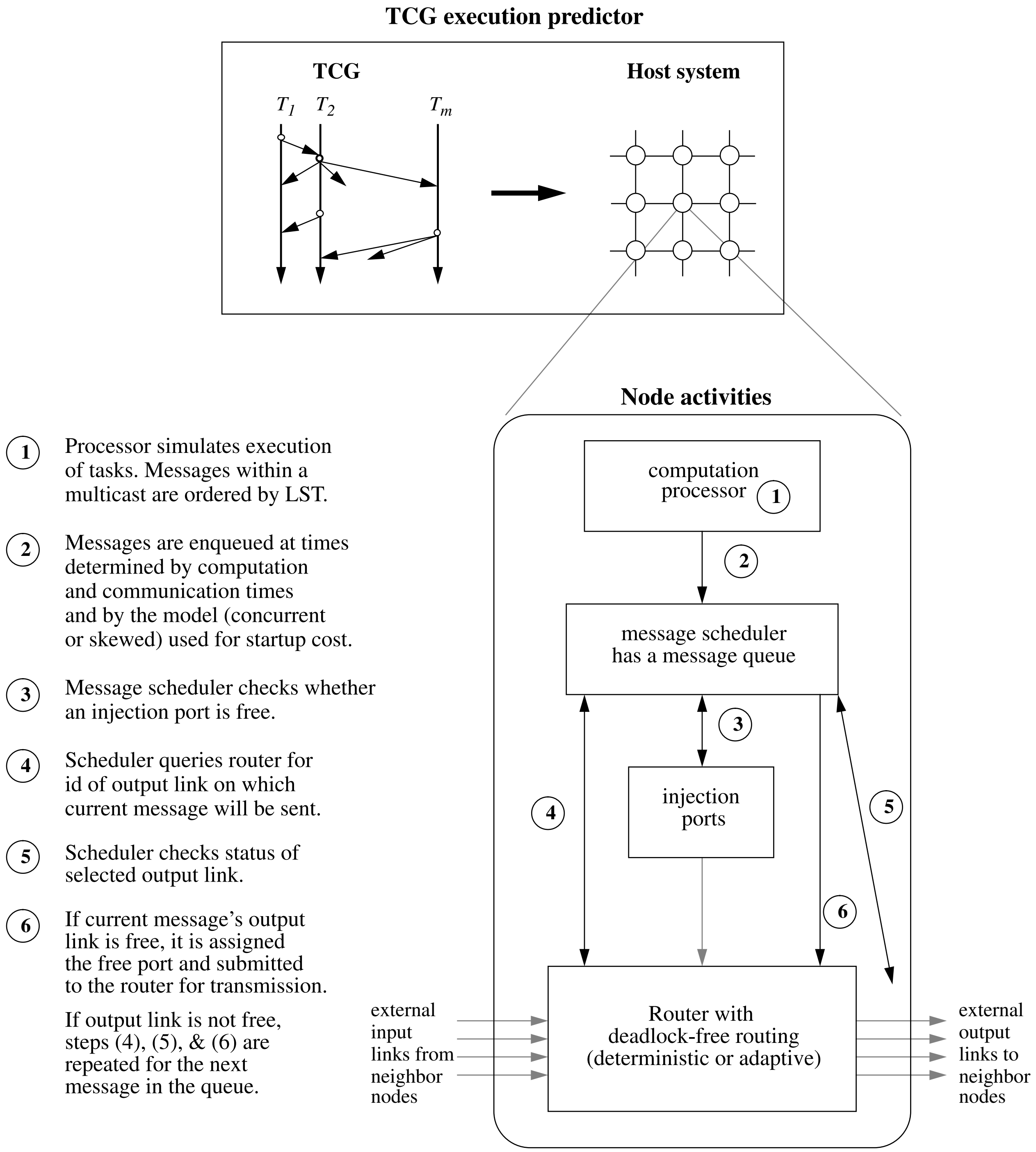

Figure 9: Dynamic message ordering algorithm.

Overtaking of the earliest message by a later message and the latter's scheduling is allowed only if (a) the earliest message's first output link is busy, (b) the later message's first output link is free, and (c) the two messages belong to the same multicast pattern.

The dynamic message ordering algorithm attempts to minimize completion time by (a) examining messages in order of their criticality (LST), and (b) reducing average message latency by improving utilization of injection ports. It is to be noted that this dynamic scheme appears to be useful only for systems having a dedicated hardware to perform the task of the message scheduler. However, it is not a must. The scheme we are proposing is quite general and can be used as a last step optimization (after clustering and task assignment) while mapping an application to a system. Since the dynamic message-ordering scheme uses dynamic state of the network, the message-ordering derived by this algorithm for each multicast communication step can be incorporated into the program for use at the run time. The program with modified message ordering will behave, with respect to links being reserved and released, almost as it would have performed in the presence of a message-scheduler. Hence, the scheme can be used on any system without having a message-scheduler hardware.

In the proposed algorithm, a later message is possibly scheduled before the earliest message in the queue only if both belong to the same multicast. However, if a hardware scheduler is available then this restriction is not necessary and ordering of messages *across* unicasts and multicasts can also be done. By restricting ordering within a multicast, a message in an LST-based multicast provided by the application will be scheduled last in its multicast pattern in the worst case. Thus, the algorithm in addition to ensuring *starvation freedom* for messages, also prevents scheduling of a message to be excessively delayed which can in fact result in increase in overall program completion time.

In the multicast pattern of Fig. 4, assume we have a 2-port system with ecube routing. Suppose the static LST-based ordering is m_0, m_{12}, m_3, m_1, m_{14}, and m_{15}. Figure 10 shows the dynamic message ordering obtained. The completion time of the multicast pattern under the three schemes, sequential, static, and dynamic is 30, 32, and 25 units, respectively. This demonstrates how the dynamic ordering algorithm can reduce completion time by examining link and port contentions. For this example, since a random LST ordering was chosen for illustration, the sequential scheme happens to outperform the static scheme. However, in general, the static scheme performs significantly better than the sequential one since LST directly reflects message criticality. The performance improvement measures are presented in the next section.

For the dynamic message ordering scheme, in terms of time complexity, the worst case occurs when messages are scheduled in the reverse order of that in the message queue. In this case, whenever a port becomes free, all messages in the queue need to be examined and the last message in the queue is scheduled. This leads to the following observation:

Observation 2 *The worst case time complexity of determining dynamic message ordering in a TCG (Temporal Communication Graph) is given by $\mathcal{O}(mcast\_num \times mcast\_deg^2)$ where $mcast\_num$ is the number of multicasts in a TCG and $mcast\_deg$ is the maximum degree, i.e. number of messages, in a multicast. This complexity is in addition to $\mathcal{O}(|V| + |E|)$ time required to determine static LST (Latest Start Time) ordering.*

However, the average time complexity for the dynamic message ordering is much less. Since the queue is ordered by message's criticality (LST) and the dynamic algorithm gives preference to critical messages (messages are examined in the order of their LST), the worst case scenario is very unlikely to happen and there are few instances of "queue probing" on an average. Our simulation experiments showed that on an average, for a multicast of degree k, the number of messages probed was less than $2k$ (as compared to k^2 in the worst case) and the number of instances of "queue jumping", i.e. number of times a later message in the multicast queue successfully overtook the first message in the queue, was between $0.05k$ and $0.24k$ (average $= 0.13k$).

Deadlock-freedom is ensured by the dynamic message ordering algorithm because the message scheduler, as described in Fig. 9, does not reserve or release any resources (external links) handled by the router. Hence, neither the message scheduler nor the dynamic message ordering algorithm introduces any additional cyclic dependencies to the underlying routing algorithm. This leads to:

Observation 3 *For a deadlock-free underlying routing scheme, the dynamic message ordering algorithm is deadlock-free.*

Assumptions:
(1) The 4x4 mesh has 2 injection ports A and B.
(2) The routing used is deterministic ecube i.e. XY routing.
(3) Ports and links are released when a message completely reaches its destination.
(4) The messages in the given multicast pattern are the only messages in the network.
(5) All messages are of length 6 flits.
(6) Startup cost model is skewed with $t_s = 1$.
(7) Latency of message m = t_s + length(m) + distance(m) - 1.

Message	Time at which message is initiated	Time at which it is scheduled	Finish Time of message	Injection port used (A or B)
Sequential message ordering				
m_0	0	1	9	A
m_1	1	9	16	B
m_3	2	9	16	A
m_{12}	3	16	25	A
m_{14}	4	16	23	B
m_{15}	5	23	30	B
		Completion Time = 30		
Static LST-based message ordering				
m_0	0	1	9	A
m_{12}	1	9	18	B
m_3	2	9	18	A
m_1	3	18	25	A
m_{14}	4	18	25	B
m_{15}	5	25	32	A
		Completion Time = 32		
Dynamic message ordering				
m_0	0	1	9	A
m_3	1	2	9	B
m_{12}	2	9	18	A
m_{14}	3	9	16	B
m_{15}	4	16	23	B
m_1	5	18	25	A
		Completion Time = 25		

Figure 10: Determining completion time of the example multicast pattern as shown in Fig. 4 under three different message ordering schemes in a 4x4 mesh.

6 Simulation Experiments and Results

We considered the Gaussian Elimination example and random task graphs in our study. We evaluated the performance of both static and dynamic message ordering algorithms with respect to reduction in program Completion Time (CT). Both the algorithms were compared against the naive sequential ordering scheme which is the default scheme used when messages in a multicast have to be sent from the source node to destination processors as a collection of unicast messages. Our study included influence of the following factors on message ordering: number of injection ports supported by the processor-router interface, degree of routing adaptivity, skewed and concurrent communication startup cost models, varying communication startup cost values, and two application classes, communication-bound and computation-bound. To eliminate the effects of clustering and assignment, we assumed a single task per processor and identity task-processor assignment scheme.

Experiments were performed using an event-driven simulator written in CSIM [24] for 16x16 and 8x8 wormhole-routed meshes. The following system parameters, representing current-generation multicomputers, were used: processor speed of 0.02 μs (microsec) per floating point operation, message communication startup time t_s of 5 μs, link propagation time t_{phy} of 5 ns (nanosec), router node delay t_{node} of 20 ns, and node switching delay t_{sw} of 5 ns. As is the case in existing architectures, we assumed that there is no separate message processor, i.e. message startup cost factor is incurred by the computation processor in addition to its contribution to message latency.

6.1 Gaussian Elimination

Figure 11 shows an explicitly-parallel code[1] and the corresponding TCG for Gaussian Elimination on an $m * m$ matrix A. For message-ordering study, we used Gaussian Elimination on a 64x64 matrix with 64 tasks and being mapped onto a 8x8 mesh. The total number of nodes and communication edges in the corresponding TCG were 2080 and 2016, respectively. The number of multicasts were 63 with the average degree of multicast being 32. The average node weight, i.e. number of floating point operations in A_{ik}, was 132 and the average communication edge weight, i.e. number of flits (1 flit = 2 bytes) in a

[1] Task T_0 consists only of A_{00} which simply sets $A(0,0)$ to 0 and broadcasts row 0 to all other tasks. Hence, without any loss of generality, we used an enhanced algorithm in which task T_0 is eliminated by assigning row 0 and row i to task T_i $(i = 1..m-1)$.

message, was 85. This gives a *Computation-to-Communication Ratio* (CCR) of (2080 * 132) / (2016 * 85) = 1.6. Thus, the Gaussian Elimination example is almost balanced with respect to computation and communication. Figures 12 and 13 show the performance of sequential, static and dynamic message-ordering algorithms for the Gaussian Elimination example for deterministic-ecube and fully-adaptive routing (algorithm described in [11] was used), respectively. Skewed model for communication startup cost with $t_s = 5\ \mu$s was assumed for both cases.

```
Task T_i (i=1..(m-1))
      /* T_i has rows 0 and i */
{
compute (A_i0);
for k = 1 to (i-1) do
      receive (Row k, T_k);
      compute (A_ik);
for j = (i+1) to n do
      send (Row i, T_j);
}
Note: A_ik represents the operation which
      sets element A(i,k) to 0 and
      updates row i as: For k+1<=j<=m-1,
      A(i,j) = A(i,j) - A(i,k) * A(k,j) / A(k,k).
```

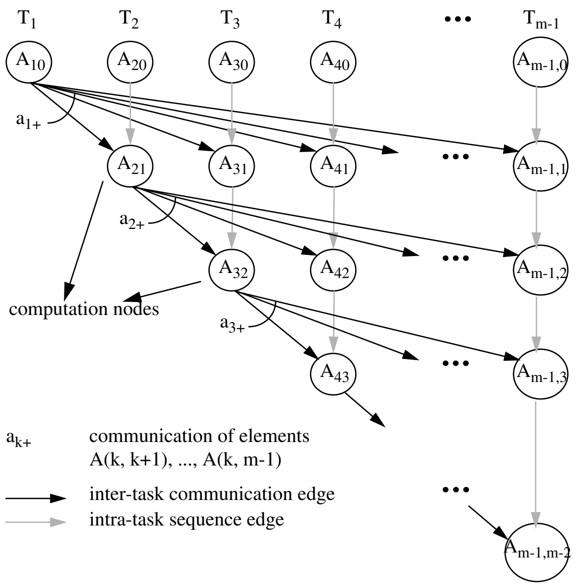

Figure 11: An explicitly-parallel program and the corresponding TCG for Gaussian Elimination on an $m * m$ matrix A.

From the two bar charts for Gaussian Elimination example, we observe that CT achieved by dynamic ordering is less than that by static which in turn is less than that by sequential ordering. Dynamic ordering reduces CT by reducing average message latency through improved utilization of injection ports. The bar charts further indicate that CT reduces as the number of injection ports increases and/or routing adaptivity increases. The percentage reduction in CT by static LST-based ordering over sequential ranges from 11% to 32.5% for varying ports and routing adaptivity. The corresponding range for dynamic over static is 10.0% to 21.9%.

With respect to routing adaptivity, the percentage reduction in CT by dynamic ordering over static is more for ecube (12.9% to 21.9%) than that for fully-adaptive (10.0% to 13.8%). This indicates that dynamic algorithm performs better for ecube as compared to fully-adaptive. This is because the static

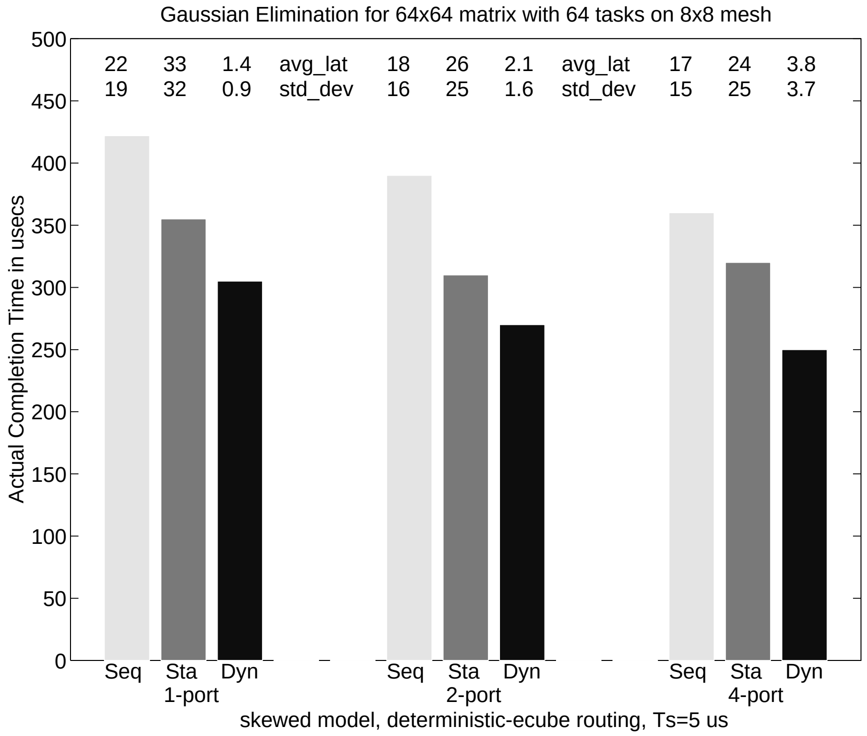

Figure 12: Comparing the three message ordering schemes for Gaussian Elimination on a 8x8 mesh with *deterministic-ecube* routing. Skewed model for communication startup cost with $t_s = 5$ μs is assumed. The two rows of numbers at the top show the mean and standard deviation of message latency with contention for the respective cases. Percentage reductions in completion time by static ordering over sequential are 15.8, 20.5, and 11.1 for 1-port, 2-port, and 4-port models, respectively. The corresponding percentages for dynamic over static are 14.1, 12.9, and 21.9.

(and also sequential) ordering scheme is able to capture some of the benefits of increased adaptivity in fully-adaptive routing leading to less improvement of dynamic scheme relative to static. Finally, with respect to the number of injection ports, the additional improvement by dynamic over static is the highest for system with 4 injection ports for both ecube (21.9%) and fully-adaptive (13.8%) routing. To summarize, the dynamic message ordering scheme is able to reduce program completion time for Gaussian Elimination considerably on systems with different routing adaptivity and number of injection ports. The best reduction is observed for ecube routed system with 4 injection ports.

An important point is to be noted regarding the performance of message ordering schemes for 1-port model. It seems that dynamic ordering must converge to the static ordering for 1-port model with ecube or fully-adaptive routing. This would indeed be the case if all links were free prior to every multicast scheduling. However, link usages in one communication phase affects the network status in

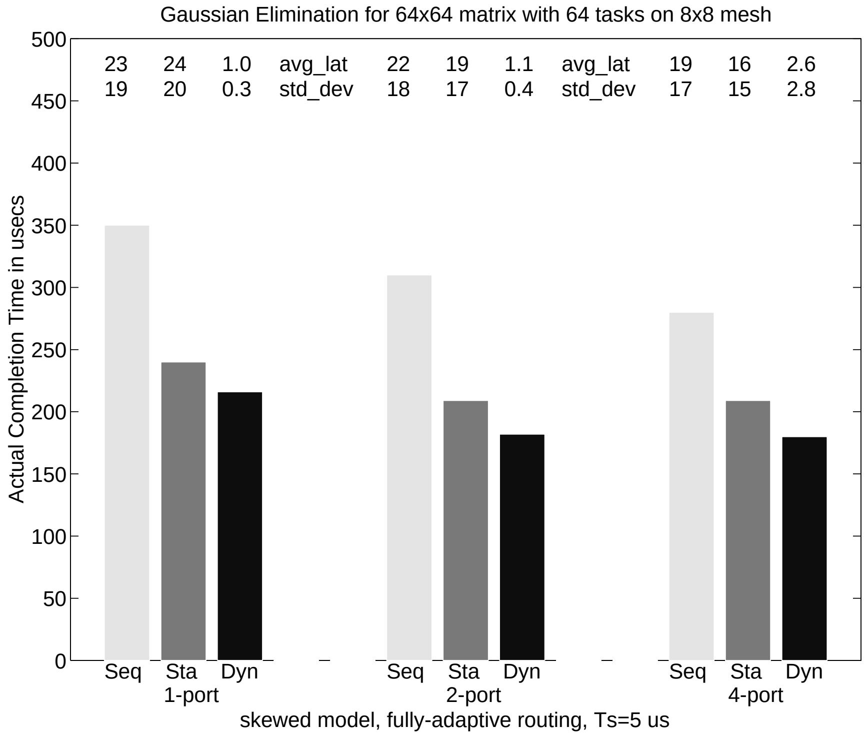

Figure 13: Comparing the three message ordering schemes for Gaussian Elimination on a 8x8 mesh with *fully-adaptive* routing. Skewed model for communication startup cost with $t_s = 5$ μs is assumed. The two rows of numbers at the top show the mean and standard deviation of message latency with contention for the respective cases. Percentage reductions in completion time by static ordering over sequential are 29.4, 32.5, and 25.3 for 1-port, 2-port, and 4-port models, respectively. The corresponding percentages for dynamic over static are 10.0, 12.8, and 13.8.

subsequent communication phases. Since the dynamic algorithm takes into account link contention when determining message ordering, it provides significant reduction in completion time over static algorithm, even for the 1-port communication model. Thus, the dynamic scheme is well suited for current-generation systems most of which support only a single port with ecube routing.

In our results, we made an interesting observation relating message latency with program completion time. The values of mean and standard deviation (shown at the top of the bar charts) of message latency with contention indicate that the dynamic scheme, compared to static, is able to reduce the average message latency by a factor of 24 for 1-port model to a factor of 6 for 4-port model. However, there is no proportionate reduction in their respective program completion times. This behavior can be explained as follows. In a parallel program, the computation phase of a task cannot begin until all its dependencies have been satisfied, i.e. until the previous computation phase of the task has completed and all the intended messages have been received. Hence, even though average message latency may be reduced, completion time may not be reduced proportionately.

In our dynamic scheme, we restricted message reordering within a multicast. However, the average latency measurements indicate that if the objective is to minimize average message latency instead of program completion time (e.g. for multicast invalidation/update messages in distributed shared memory systems), then ordering of messages within and *across* unicasts and multicasts may possibly be beneficial.

6.2 Random Graphs

Experiments were performed on 100 random TCGs of two classes each: communication-bound with CCR of 0.3 and computation-bound with CCR of 3.0. Details on random TCG generation can be found in our earlier work [9, 10]. For 16x16 meshes having 256 communication-bound tasks, the degree of multicast was chosen to be an exponential distribution with mean 16. The mean for 8x8 meshes with 64 computation-bound tasks was 3. Figures 14 to 17 summarize the performance of the three message ordering schemes, sequential, static, and dynamic for (a) 1-port and 4-port models, (b) ecube and fully-adaptive routing, (c) skewed and concurrent communication startup cost models, (d) communication startup cost t_s = 30, 15, 5, and 1 μs, and (e) communication and computation-bound TCGs. The following observations can be made from these simulation results.

The static LST-based message ordering reduces program CT by 9.8% to 33.1% as compared to naive sequential ordering. This shows that overall program CT can be significantly reduced by taking into account criticality of messages in an application. The dynamic message ordering scheme provides an *additional* reduction in completion time by 7.1% to 17.7%. This indicates that by taking into account link contention, port contention, and routing adaptivity, further reduction in program CT can be achieved.

Effect of number of injection ports: For all the bar charts shown in Figs. 14, 15, and 17, CT reduces as the number of injection ports supported by the processor-router interface is increased from 1 to 4. The percentage reduction in CT by static over sequential is most often more for 1-port model (16.5% to 30.3%) than that for 4-port model (9.8% to 18.7%, 33.1% in one case). This is because the sequential ordering scheme itself is able to take advantage of multiple ports.

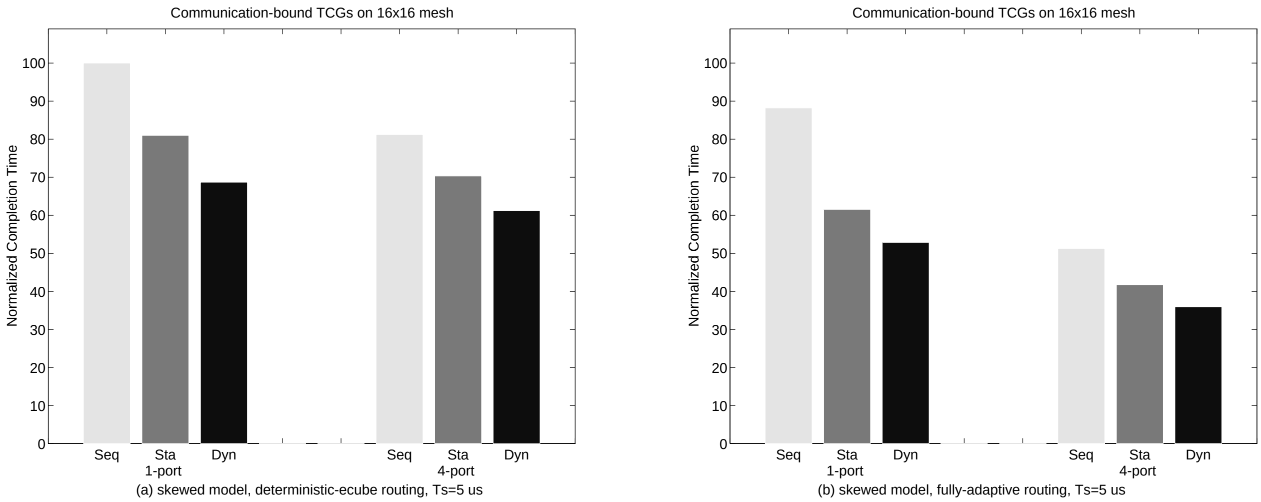

Figure 14: Comparing normalized Completion Time (CT) of *communication-bound* TCGs with the three message ordering schemes on a 16x16 mesh. *Skewed* model for communication startup cost with $t_s = 5$ μs is assumed. (a) Routing is *deterministic-ecube*. Percentage reductions in CT by static ordering over sequential are 19.0 and 13.4 for 1-port and 4-port models, respectively. The corresponding percentages for dynamic over static are 15.2 and 12.9. (b) Routing is *fully-adaptive*. The percentage reduction values for 1-port and 4-port model are 30.3 and 18.7 for static over sequential and 14.1 and 13.9 for dynamic over static. CTs in (a) and (b) are all normalized with respect to CT of 1-port sequential ordering in (a).

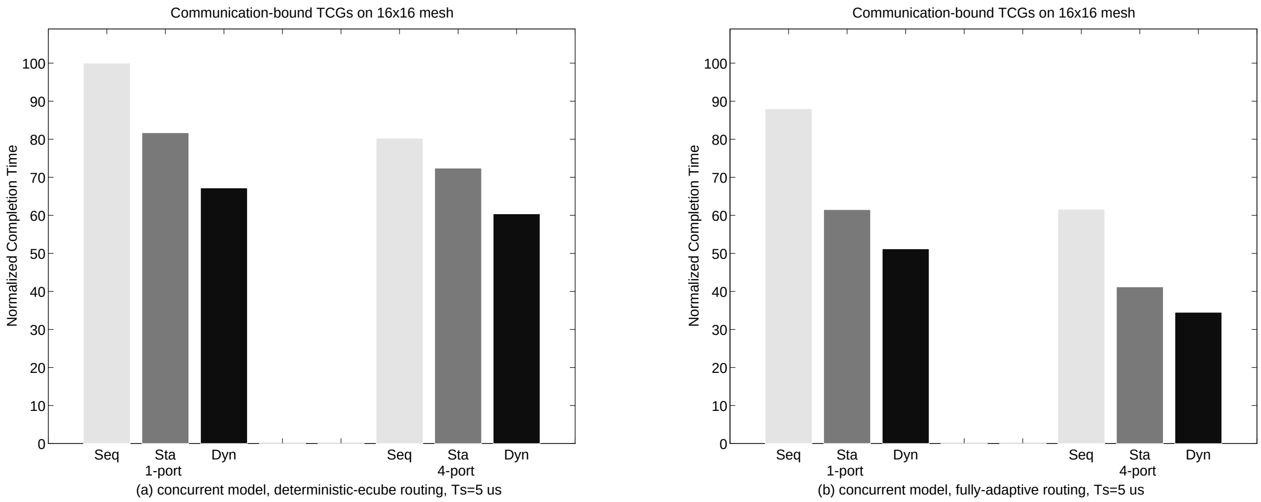

Figure 15: Comparing normalized Completion Time (CT) of *communication-bound* TCGs with the three message ordering schemes on a 16x16 mesh. *Concurrent* model for communication startup cost with $t_s = 5$ μs is assumed. (a) Routing is *deterministic-ecube*. Percentage reductions in CT by static ordering over sequential are 18.3 and 9.8 for 1-port and 4-port models, respectively. The corresponding percentages for dynamic over static are 17.7 and 16.6. (b) Routing is *fully-adaptive*. The percentage reduction values for 1-port and 4-port model are 30.1 and 33.1 for static over sequential and 16.7 and 16.3 for dynamic over static. CTs in (a) and (b) are all normalized with respect to CT of 1-port sequential ordering in (a).

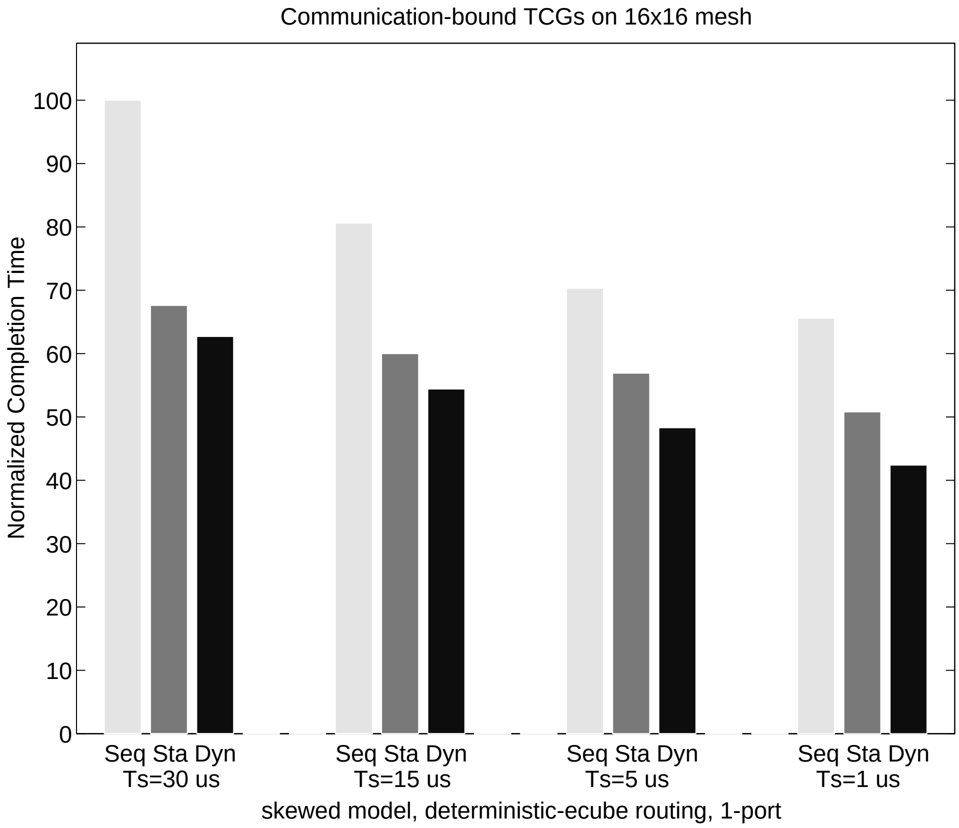

Figure 16: Comparing normalized Completion Time (CT) of *communication-bound* TCGs with the three message ordering schemes on a 16x16 mesh with *varying t_s values*. *Skewed* model for communication startup cost, deterministic-eube routing, and 1-port model are assumed. Percentage reductions in CT by static ordering over sequential are 32.4, 25.6, 19.1, and 22.6 for t_s = 30, 15, 5, and 1 μs, respectively. The corresponding percentages for dynamic over static are 7.2, 9.4, 15.2, and 16.5.

For communication-bound TCGs (Figs. 14 and 15), the percentage reduction in CT by dynamic over static is more for 1-port model (14.1% to 17.7%) than that for 4-port model (12.9% to 16.6%). As mentioned earlier for Gaussian Elimination example, link usages in a multicast communication phase can cause link contentions in subsequent communication phases. Similarly, multicast messages sent by a processor in a communication phase can cause link contention for another processor's multicast in the same phase. As a result, the performance of dynamic message ordering is not same as that of static ordering for 1-port model.

For computation-bound TCGs (Fig. 17), the percentage reduction in CT by dynamic over static is more for 4-port model (10%) than that for 1-port model (7.1%). Since computation-bound TCGs have larger computation phases, there is an increased chance of links being free prior to a multicast

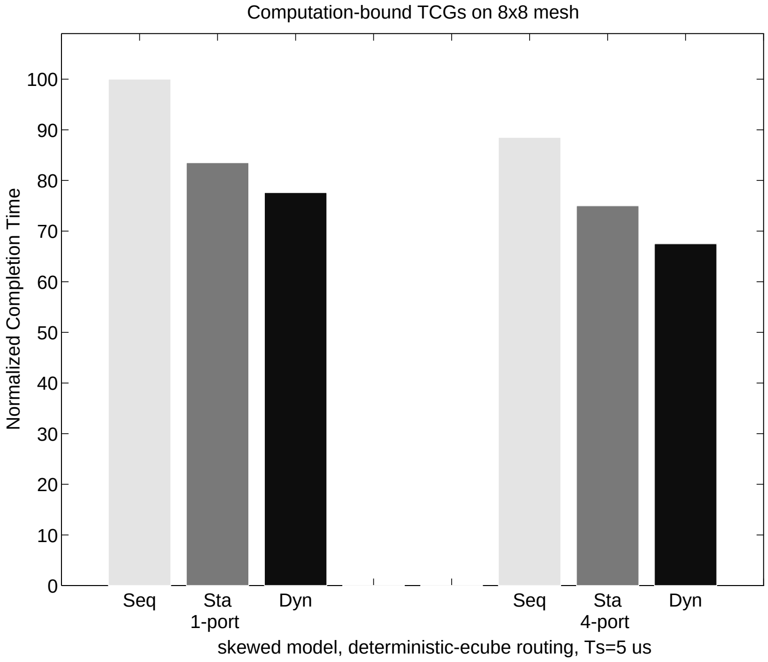

Figure 17: Comparing normalized Completion Time (CT) of *computation-bound* TCGs with the three message ordering schemes on a 8x8 mesh. *Skewed* model for communication startup cost with $t_s = 5$ μs and deterministic-ecube routing are assumed. Percentage reductions in CT by static ordering over sequential are 16.5 and 15.3 for 1-port and 4-port models, respectively. The corresponding percentages for dynamic over static are 7.1 and 10.0.

communication. As a result, the dynamic scheme performs better with increase in number of injection ports.

Effect of routing adaptivity: For the bar charts shown in Figs. 14 and 15, CT reduces as the routing adaptivity is increased from deterministic (least adaptive) to fully-adaptive. The percentage reduction in CT by static over sequential is more for fully-adaptive routing (18.7% to 33.1%) than that for ecube routing (9.8% to 18.3%). The percentage reduction in CT by dynamic over static is almost the same for both ecube (12.9% to 17.7%) and fully-adaptive routing (13.9% to 16.7%). Ideally the dynamic scheme must be able to further improve message ordering by finding alternate output links due to increased adaptivity. However, since the static scheme itself captures some of the benefits of increased adaptivity, the dynamic scheme shows relatively less improvement than expected.

Effect of communication startup cost model: Clearly, given an application, CT for concurrent model will be less than that for skewed model. For concurrent model, only a single startup cost t_s per multicast is included. For skewed model, startup cost is encountered for each message in a multicast.

The bar charts shown in Figs. 14 and 15 show the normalized CTs for skewed and concurrent startup models, respectively. The percentage reduction in CT by dynamic over static is more for concurrent (16.6% to 17.7%) than that for skewed (12.9% to 15.2%) model. This is because with concurrent model, all messages in a multicast are ready simultaneously and this introduces more choices for the dynamic algorithm in selecting messages for scheduling.

Effect of communication startup cost t_s: The bar chart in Fig. 16 shows the effect of varying t_s values on the three message ordering schemes. Skewed model, ecube routing, and 1-port model were assumed. As t_s value reduces from 30 μs to 1 μs, the percentage reduction in CT by dynamic over static increases (from 7.2% to 16.5%). This is again due to the simultaneous availability of more messages as t_s value reduces. Hence, as startup cost values continue to reduce for new generation architectures, the dynamic message ordering scheme will be increasingly beneficial in reducing program completion time.

Effect of Computation-to-Communication Ratio: The bar charts in Figs. 14 to 16 are for communication-bound TCGs and the chart in Fig. 17 is for computation-bound TCGs. Both static and dynamic schemes provide more percentage reduction in CT for communication-bound TCGs than for computation-bound TCGs. This is obvious since it is only message communication cost and not computation cost factor in completion time that can possibly be reduced.

7 Conclusions

This study indicates that message-ordering plays a significant role in determining program completion time in distributed-memory systems. We have proposed message-ordering algorithms that consider temporal information of task computation and communication, criticality of messages, and dynamic system state such as link and port contentions to reduce program completion time. The proposed dynamic message-ordering algorithm is shown to perform far better than the sequential- and static-ordering algorithms. The reductions are shown to be significant particularly for systems having one injection port, supporting ecube or adaptive routing, and applications being communication-bound. It indicates that many applications running on the current generation wormhole-routed systems can take advantage of the proposed dynamic message-ordering algorithms to reduce program execution time.

The results presented in this paper are based on Gaussian Elimination example and random task graphs. We are continuing our work to apply the scheme to other scientific and numerical applications. We are also exploring to use the message-ordering algorithm in developing optimal communication algorithm for multiple multicast pattern, used for data redistribution between computation phases, on wormhole-routed systems. The current generation systems are increasingly using new architectural trends like multithreading and having separate communication processor. It will be interesting to study the impact of message-ordering on systems with these new trends.

References

[1] S. Balakrishnan and D. K. Panda. Impact of Multiple Consumption Channels on Wormhole Routed k-ary n-cube Networks. In *International Parallel Processing Symposium*, pages 163–167, 1993.

[2] S. W. Bollinger and S. F. Midkiff. Heursitic Technique for Processor and Link Assignment in Multicomputers. *IEEE Transactions on Computers*, 40(3), Mar. 1991.

[3] V. Chaudhary and J. K. Aggrawal. A Generalized Scheme for Mapping Parallel Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):328–346, Mar. 1993.

[4] A. A. Chien and J. H. Kim. Planar Adaptive Routing: Low-cost Adaptive Networks for Multiprocessors. In *19th Annual International Symposium on Computer Architecture*, pages 268–277, May 1992.

[5] Cray Research, Inc. *Cray T3D System Architecture Overview*, 1993.

[6] W. J. Dally. Virtual-channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3:194–205, Mar. 1992.

[7] S. Darbha and D. P. Agrawal. A Task Duplication Based Optimal Scheduling Algorithm for Variable Execution Time Tasks. In *International Conference on Parallel Processing*, pages II:52–56, Aug 1994.

[8] M. Dikaiakos, A. Rogers, and K. Steiglitz. Message Ordering in Multiprocessors with Synchronous Communication. In *International Conference on Parallel Processing*, volume III, pages 196–203, 1992.

[9] Vibha A. Dixit-Radiya and Dhabaleswar K. Panda. Task Assignment on Distributed-Memory Systems with Adaptive Wormhole Routing. In *Symposium on Parallel and Distributed Processing*, pages 674–681, 1993.

[10] Vibha A. Dixit-Radiya and Dhabaleswar K. Panda. Clustering and Intra-Processor Scheduling for Explicitly-Parallel Programs on Distributed-Memory Systems. In *International Parallel Processing Symposium*, pages 609–615, 1994.

[11] J. Duato. Deadlock-Free Adaptive Routing Algorithms for Multicomputers: Evaluation of a New Algorithm. In *Symposium on Parallel and Distributed Processing*, pages 840–847, Dec. 1991.

[12] 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.

[13] D. Lenoski et. al. The Stanford DASH Multiprocessor. *IEEE Computer*, pages 63–79, Mar. 1992.

[14] P. McKinley et al. Unicast-Based Multicast Communication in Wormhole-Routed Networks. In *International Conference on Parallel Processing*, volume II, pages 10–19, 1992.

[15] Vipin Kumar et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Inc., 1994.

[16] A. Gerasoulis and T. Yang. A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[17] J. L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[18] Intel Corp. *Paragon XP/S Product Overview*, 1991.

[19] A. Kumar, S. Ramakrishnan, C. Deshpande, and L. Dunning. Performance Comparison of Two Algorithms for Task Assignment. In *International Conference on Parallel Processing*, pages III:83–87, Aug 1994.

[20] N. Mehdiratta and K. Ghose. A Bottom-up Approach to Task Scheduling on Distributed Memory Multiprocessors. In *International Conference on Parallel Processing*, pages II:151–154, Aug 1994.

[21] L. M. Ni and P. K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pages 62–76, Feb. 1993.

[22] D. K. Panda, S. Singhal, and P. Prabhakaran. Multidestination Message Passing Mechanism Conforming to Base Wormhole Routing Scheme. In *Proceedings of the Parallel Computer Routing and Communication Workshop*, 1994. To appear in Lecture Notes in Computer Science, Springer-Verlag.

[23] Dhabaleswar K. Panda and Vibha A. Dixit-Radiya. Message-Ordering for Wormhole-Routed Multiport Systems with Link Contention and Routing Adaptivity. In *Scalable High Performance Computing Conference*, pages 191–198, 1994.

[24] H. Schwetman. Introduction to Process-Oriented Simulation and CSIM. In *Proceedings of Winter Simulation Conference*, 1990.

[25] B. Shirazi, M. Wang, and G. Pathak. Analysis and Evaluation of Heuristic Methods for Static Scheduling. *Journal of Parallel and Distributed Computing*, 10:222–232, 1990.

[26] G. C. Sih and E. A. Lee. A Compile-time Scheduling Heuristic for Interconnection-contrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, Feb. 1993.