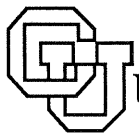
Dynamic Load Distribution on Point-to-Point Multicomputer Networks

Dirk C. Grunwald
Bobby A.A. Nazief
Daniel A. Reed

CU-CS-542-91 August 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Dynamic Load Distribution on Point-to-Point Multicomputer Networks

Dirk C. Grunwald
Bobby A. A. Nazief
Daniel A. Reed

CU-CS-542-91 August 1991

University of Colorado at Boulder

Technical Report CU-CS-542-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Dynamic Load Distribution on Point-to-Point Multicomputer Networks*

Dirk C. Grunwald†
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

Bobby A. A. Nazief ‡ Daniel A. Reed ‡
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

August 1991

Abstract

To benefit from parallel computers, programs must be partitioned into units that work in parallel. Once partitioned, these units, called processes, tasks or threads, must be assigned to specific processors for execution. On shared memory architectures, this is termed *scheduling*, whereas on multicomputer systems, it is called *load distribution*, distinguishing it from any local scheduling used on individual nodes. Load distribution algorithms determine the *initial placement* for tasks, a precursor to the more general problem of *load redistribution.*

We compared several load placement algorithms using both instrumented programs and synthetic program models. Salient characteristics of these program traces (e.g., total computation time, total number of messages sent and average message size) span two orders of magnitude. We simulated a modern architecture with point-to-point communication. To understand the interactions of communication network characteristics, number of processors, and workload, we analyzed the dynamics of task placement using measures of both the hardware utilization and the placement algorithm behavior. We found that information is usually better than inference for driving process placement, but that informationless strategies often are equal or superior to both. The strategies we examined use load or status information to select placement locations; this information is explicitly disseminated or is piggybacked on normal communication. We also found that extant point-to-point networks reduce the rate of information dissemination because transiting messages are ignored by intermediate nodes. From these studies, we have concluded that the relative performance of placement strategies depends on the structure of the task creation tree. For shallow process trees, desirable workload distribution strategies will place new processes globally, rather than locally, spreading processes rapidly.

1 Introduction

Parallel computation offers surcease for computational limitations when solving larger or more complex problems than is possible on single processor systems. Of the many varieties of possible

*An abridged version of this paper appeared in the *Proceedings of the Fifth Distributed Memory Computing Conference*, April 8–12, 1990, Charleston, NC.

†Supported in part by the National Science Foundation under NSF Grant CCR-9010624.

‡Supported in part by the National Science Foundation under grants NSF CCR86-57696, NSF CCR87-06653 and NSF CDA87-22836, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, and by a grant from the Digital Equipment Corporation External Research Program.

parallel computer architectures, we consider one architecture known as the *multicomputer network* or simply multicomputers [29]. Multicomputers consist of a large number of interconnected computing nodes that asynchronously cooperate via a message passing network to execute the tasks of parallel programs. Each computing node minimally contains a processor, a local memory, and a communication controller.

Multicomputers have evolved from machines with standard computation nodes and low-speed communication networks to current generation systems with high-performance computation nodes and high-speed communication networks. First generation multicomputers (e.g., the Ametek System/14 [1], the Intel iPSC/1 [28], and the Ncube/ten [17]) used computational nodes whose peak performance was comparable to early personal computers. Moreover, their communication network performance was poor; *store-and-forward* packet switching created communication latency proportional to the distance between the source and destination nodes.

The processing speed of second generation multicomputers (e.g., the Symult 2010 [30] and the Intel iPSC/2 [2]) increased by a factor of four. More importantly, they provided lower latency, circuit-switched communication networks. The improvement in both computation and communication performance is even more striking in the coming generation of multicomputers (e.g., the Intel Touchstone system [23]). Node computation speed is expected to increase by an additional order of magnitude and communication performance should increase by two orders of magnitude.

In contrast to revolutionary hardware changes, multicomputer software has evolved slowly. Traditional programming languages such as C and Fortran, coupled with library support for message passing, still are used to program distributed memory parallel systems. The high communication latency of first generation systems, coupled with simple software tools, placed a heavy intellectual burden on application program developers; it was not possible to write application programs without intimate knowledge of the machine architecture.

Better communication hardware in second generation machines eased the programming burden. Communication locality remained important, but its absence was not debilitating. Concurrently, newly created research software allowed the construction of parallel programs without intimate knowledge of the machine architecture; examples include Cantor [3], Linda [5], the Chare Kernel [20], and Mentat [12]. In general, these tools provide an abstraction of a task that can be dynamically created or destroyed, and they support a programming abstraction where the intertask communication pattern is independent of task location. Moreover, both the number of tasks and the pattern of intertask communication can vary during program execution. To benefit from the parallelism offered by multicomputers, these tasks must be dynamically assigned to specific nodes for execution. On shared memory architectures, this is termed *scheduling*; on multicomputer systems, it is called *load distribution* to distinguish it from the local scheduling algorithm used on individual nodes.

In this paper, we explore the automatic distribution of dynamically created tasks in low-latency, point-to-point communication networks. We compare policies for *load placement*, rather than the more general problem of *load redistribution*. Load placement strategies place tasks when they are created, whereas load redistribution strategies rearrange extant tasks. Several groups have proposed load placement strategies [22, 32, 25, 19, 31, 4, 34, 33], yet few comparative studies exist. Fewer studies exist for modern, low-latency network architectures.

The objective of task placement can be succiently stated as follows. Given a parallel program whose dynamically created tasks communicate via message passing, assign these tasks to multicomputer nodes to maximize parallelism and minimize program execution time. The simplicity of the

problem description belies its difficulty. Assigning tasks to nodes requires knowledge of the time-varying system state — acquiring this information is not without cost. Any placement strategy must balance the volume of state information against both its accuracy and cost; as the number of multicomputer nodes increases, maintaining an accurate, distributed view of the system state becomes problematic.

In §2, we begin by reviewing the possible approaches to dynamic task placement and their relation to the application program's computational model — the pattern of task interactions delimits the range of feasible task placement strategies. Based on this schema, §3–§4 summarize the placement strategies and application program workloads we selected for study. The latter include both synthetic and captured task creation patterns and represent both highly computation and communication intensive programs. Salient characteristics of these programs (e.g., total computation time, total number of messages sent and average message size) span two orders of magnitude. Although static performance measures, described in §5, suggest the efficacy of particular placement strategies, they cannot reveal the underlying reasons. Understanding these reasons requires a careful analysis of system dynamics; this analysis is the subject of §6. Finally, §7 summarizes our observations and offers suggestions for future work.

2 Review of Prior Work

Although many task distribution heuristics have been proposed, most studies have adopted differing hardware models or have assumed disparate computational models. Because the efficacy of task distribution heuristics is inextricably tied to the underlying assumptions, we begin by examining a spectrum of possible computational models, followed by a review of archetypical task placement heuristics.

2.1 Computational Models

Earlier, we alluded to programming tools that hide machine architecture idiosyncrasies from software developers. Many such tools support the development of programs that consist of dynamically created tasks that must be distributed across processors during program execution. Both dynamic task creation and the associated intertask communication can be realized in many ways. A particular combination of dynamic task creation and intertask communication patterns delimits a subset of computational models.

In the simplest model, all tasks are created at the beginning of the computation and compute independently with no intertask communication. In an extension of this model, each task communicates only with a fixed subset of the other tasks. Thus, the pattern of intertask communication is regular in both space and time. This computational model was often used in programs for first generation hypercubes. A classic example is a simple, iterative partial differential equations solver, where each task communicates with four other tasks in a grid communication pattern.

When task creation occurs more than once (i.e., when a task can be created at any time during program execution) the computational model becomes more complex, and the spatial pattern of intertask communication becomes dynamic. It can be simplified by restricting the possible types of intertask communication. As an example, one might restrict communication to a task and its children. Communication can be further restricted to task creation, as the *invocation* message sent

by a task to its child, and at child's termination, as the result returned by the child to its parent. This could be interpreted as a computational model of functional programming.

We assume that computation on a multicomputer is performed by tasks that communicate via messages. We have modeled their execution behavior by *dynamic flowgraphs* that describe the time-varying relationships of processes and their interactions [13]. A *task* is an abstraction for a persistent state associated with a computation. It may send and receive messages, and it has a finite lifetime with an explicit times of invocation and termination. An *action* is the execution by a task of a series of computation statements. The computation does not rely on external resources, such as communication with another task. Actions correspond to the intervals between task invocations and message operations; during these periods, the task is computing. A task is delayed by a *communication dependency* if it requires information from another task. Until that information is received, the task cannot continue execution. A *dynamic flowgraph* is a dynamic sequence of actions and communications representing a task or group of interacting tasks. The dynamic flowgraph depicts all tasks, actions and communication dependencies, but not scheduling or distribution decisions. This representation excludes programs that react to the system state, such as programs for real-time control.

A dynamic flowgraph is a *dynamic flowtree* if all non-root tasks outlive their children and tasks communicate only with their children. A *pure dynamic flowtree* is a dynamic flowtree where tasks receive messages only when they born or when a child dies, and they send messages only when they complete or when creating children. Of the workloads described in §4, most tend to be dynamic flowtrees, although a considerable fraction are the more restricted pure dynamic flowtree.

2.2 Load Distribution Studies

There have been many studies of load placement. Below, we survey some of the experimental methods proposed; these are notable because either they are archetypical, or they have been the subject of simulation studies.

2.2.1 Diffusion Scheduling

In one of the earliest proposals, Sullivan suggested *diffusion scheduling* for workload distribution in the CHoPP system [32], a binary hypercube of processors. At the time of task creation, a *task invocation* message is broadcast to a subcube of the nodes within the network. Idle nodes bid on the request, and one such bid is chosen by the originating node. If no node is idle, the last request of the nearest requester is retained. When busy nodes become idle, they bid on the previous request. When an originating node selects a bid from the set of respondents, it broadcasts a cancellation message to all nodes with losing bids.

In the CHoPP scheme, task distribution is localized, but it allows a gradual diffusion of work throughout the network. One can estimate the number of messages needed to distribute a task as follows. Assume the broadcast is to a k-dimensional subcube, and the probability of a node being busy is P_{busy}. In practice, P_{busy} is the mean utilization of all system nodes. We assume that query messages are not propagated by idle nodes because they tender a bid for the task.

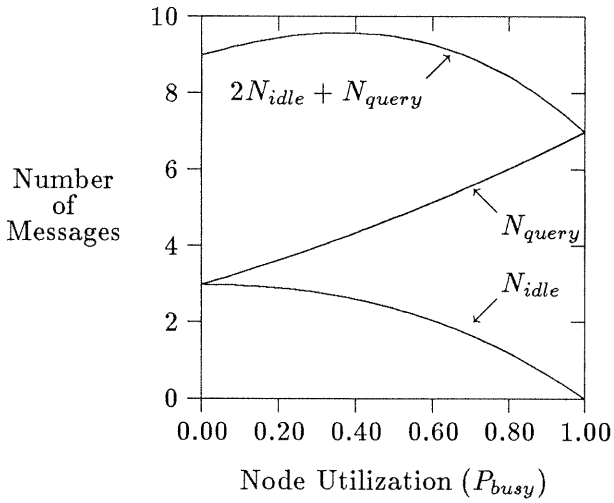


Figure 1: Number of Messages Sent by Diffusion Scheduling in a Binary 3-Subcube

In a binary hypercube topology, there are $\binom{k}{i}$ nodes exactly i hops distant in a binary k-subcube. The number of nodes that receive messages in the initial query is

$$N_{query} = \sum_{i=1}^{k} \binom{k}{i} P_{busy}^{i-1}. \tag{1}$$

Each of the $\binom{k}{i}$ nodes receives the query if all $i-1$ nodes along its broadcast path are busy; the probability of the latter is P_{busy}^{i-1}. Of the N_{query} nodes that receive a bid request,

$$N_{idle} = N_{query}(1 - P_{busy}), \tag{2}$$

are idle, and respond with a bid.

Cancellation messages are of the form "the bid from node j was accepted," and nodes that were previously busy forward the cancellation message. Because idle nodes do not forward the original request, forwarding the cancellation is unnecessary; there can be no nodes waiting "downstream" for the cancellation. Our analysis does not consider the limiting case where *no* idle node exists. In this case, the cancellation message is delayed until at least one node tenders a bid for the task. It is possible that all nodes will respond to the bid before the cancellation arrives.

The lower bound for the number of messages sent for a single query is $2N_{idle} + N_{query}$. Recall that we are broadcasting in a binary k-subcube of the network, where k is the diameter of the subcube containing 2^k nodes. Although $2N_{idle} + N_{query}$ total messages are sent, broadcast uses a divide-and-conquer algorithm, and the *originating node* sends at most $2k$ messages: k for the query and k for cancellation. The other messages are generated by intermediate nodes. However, the originating node receives and processes a message from *each* idle node, or N_{idle} messages; this is the bottleneck in the diffusion strategy.

Figure 1 shows N_{idle} and N_{query} when scheduling within a binary 3-subcube containing eight nodes. Recall that we broadcast to a subcube, or portion, of a network. At fifty percent processor utilization, approximately ten messages would be sent to schedule a *single* task; on average, two of the four idle nodes would respond.

Thus, broadcast schemes can locate an idle node in a subcube by sending a large number of messages. We will see that other methods can locate the same node using fewer messages. Moreover, the performance of broadcast schemes is poorer than that of other, simpler methods. For this reason, we have not considered complex load placement policies, such as CHoPP diffusion scheduling.

2.2.2 Gradient Scheduling

The Rediflow multiprocessor system [21] was based on a mesh-connected network of nodes. For this system, Lin [24, 25] proposed the *gradient strategy*, a local, demand driven, adaptive load sharing strategy for task scheduling. This strategy, and others based on it, can be termed *limited diffusion* strategies, to differentiate them from the diffusion strategy of CHoPP. In limited diffusion strategies, task invocation requests are sent through adjacent nodes as with diffusion scheduling. However, more information is maintained regarding the system state, and this information is used to limit the number of messages needed to distribute a task. This reduces the overhead of the load distribution strategy and increases the stability of the distribution system.

In Lin's strategy, tasks represent pure function invocations with no other communication. When a task in created, it will be placed at the closest possible idle node. Ideally, any tasks that communicate with the new task will likewise be placed at nodes close to the new node, because they will be children of the new task. Thus, the gradient strategy seeks to exploit the limited communication locality of functional programs.

Nodes are assigned a *pressure*, a function of both processor utilization and memory availability. The pressure is used to divide nodes into three categories: idle, neutral or abundant. Each node also defines a *proximity function*, indicating the distance to the nearest idle node. The proximity function is computed using the pressure of each node and the proximity function of adjacent nodes. The value of the proximity function at each node forms a distributed *gradient plane*. Nodes with lower proximity are closer to nodes with a low pressure. In general terms, valleys represent regions of low pressure; nodes in that area can accept additional work. Each node broadcasts its proximity value to adjacent nodes using *status messages* that play the same role as query messages in diffusion scheduling. The flow of status messages is adaptively damped in a steady-state system, when pressure at each node is constant.

Lin shows that the minimum length path to an idle node is found by following the trail of least proximity functions. When a new task is created, the task invocation message follows the gradient plane until it reaches an idle node. If an idle node accepts a task and is transformed to a neutral state, the gradient plane is updated to reflect this change. A limit on the number of hops a task invocation message can travel prevents possible livelock. Otherwise, an invocation message could pursue an elusive, shifting low point in the gradient plane. Lin states that task invocation messages always move in the direction of the currently least loaded node. In practice, this is only true if every node has the true status of adjacent nodes; the invocation message may be misdirected by stale information.

Gradient strategies depend on strictly local communication, and they are adaptable to a variety of interconnection networks while also tolerating node failures. By inferring the global system state using information from adjacent nodes, load distribution can be achieved without the large number of scheduling messages required for pure diffusion scheduling. In Rediflow simulations, Lin determined that the gradient strategy performs significantly better than no load balancing. Lin

also compared the gradient strategy to a central scheduler that uses load information from all nodes to schedule tasks; he found that the performance approached that of the central scheduler when a sufficiently small status update interval was used.

2.2.3 Contracting Within a Neighborhood

Others have studied strategies based on the idea of limited diffusion or gradient planes. Kalé [18] proposed a strategy called *Contracting Within A Neighborhood* (CWN) for placement of individual clauses in a parallel Prolog system. It was compared to Lin's gradient plane strategy in a study by Kalé [19] and a thesis by Carroll [6]. Kalé [19] determined that the gradient strategy did not distribute tasks rapidly enough in an idle system. Thus, CWN imposes minimum and maximum forwarding thresholds, forwarding a task at least once, but no more than a predetermined limit. Invocation requests are always sent to the topologically adjacent node with the least load; this differs from the gradient strategy, where invocation requests are sent in the *direction* of an inferred global minimum. Sending invocation requests to the node with the least load can induce a transient local minimum.

Shu and Kalé [31] have recently developed the *Adaptive* Contracting Within a Neighborhood (ACWN) distribution strategy. To distinguish the two, they refer to the CWN strategy previously presented as *Naive* CWN (NCWN). The ACWN strategy is similar to NCWN; however, it adapts to varying loads by dynamically altering the minimum and maximum distance a task invocation message travels. Shu and Kalé have implemented this algorithm on an Intel iPSC/2 hypercube and found that it results in shorter execution time than either the gradient strategy or random placement.

2.2.4 Random Task Placement

Athas and Seitz [4] have proposed *global, random task placement*; a random node is selected for each invocation request. Random allocation is appealing because of its implementation simplicity; moreover, issues of event horizons, stale information and livelock are obviated. Although random placement appears obvious, it was only recently proposed in earnest. Global random placement ignores communication locality. In store-and-forward networks, this greatly increases communication overhead. In circuit-switched or wormhole networks, communication locality is less important, making random placement feasible.

Eager *et al.* [10] have examined *load redistribution*, a problem similar to load placement. An interesting conclusion from their study was that *any* redistribution strategy was better than none, and that *simple* policies were almost as effective as more complex ones. Although the problems differ, this offers encouragement for simple strategies such as random placement.

3 Load Distribution Strategies

Based on our survey of task placement research, we call a measured value describing the "load" of a processor the *load metric* whereas a *status metric* is a synthesized value combining the measured load value of a processor with other information. Examples of load metrics are the total number of tasks, the number of tasks ready for execution or processor utilization. A status metric usually indicates the current or predicted load of a subset of the multicomputer nodes. Load distribution

Network	Bandwidth	Message Header Size	Node Latency	Switch Latency
Normal	32 Mbytes/second	32 Bytes	10 microseconds	1 microsecond
Slow	32 Mbytes/second	32 Bytes	100 microseconds	1 microsecond

Table 1: Hardware Characteristics of Simulated Networks

strategies can use either or both metrics when placing processes. Certain strategies also limit the *source* of such metrics, using only local or regional information to make scheduling decisions, whereas others attempt to use information from the entire network.

We have compared five *families* of load distribution policies. Within each policy family, we examined several strategies that use different decision metrics (either load, status or no metric) and different sources of information (either adjacent nodes, nodes within a fixed radius or the entire network). Each strategy is purposefully simple to highlight the potential effectiveness of each.

We implemented the different strategies using a simulated binary hypercube network with performance comparable to the CalTech/JPL Mark-IIIe HyperSwitch network [8]. The HyperSwitch network is a point-to-point, circuit-switched network. Thus, only the source and destination nodes for each message directly interact with those messages. Messages use a crossbar switch to *cut-through* intermediate nodes, reducing the overhead for non-local communication. Each node has a communication processor, freeing the main computation processor from mundane communication tasks. The Intel iPSC/2 has a similar network; however, it has no communication processor and has much higher communication latency. We simulated two networks, with the parameters shown in Table 1. The "normal" network represents an aggressive network architecture, whereas the "slow" network is more representative of implementations that have non-trivial software latency (e.g., due to message buffer management).

We implemented a simulacrum operating system; it has enough substance to schedule processes, orchestrate message delivery and implement a system scheduling task. Processes in an application program are represented by tasks in the simulated system. Each task is assigned to a specific node in the simulated system by one of our load distribution strategies. Tasks at the same node compete for the single computation processor (CPU) at that node. Within a node, the CPU uses a first-come-first-serve scheduler with two millisecond timeslices. We assume that the load distribution strategies are implemented by the communication processor.

A task placement strategy contains two important, interacting components, namely information acquisition and decision policy. Using the acquired information, the decision policy selects network regions or individual nodes for potential task placement; we briefly review both information sources and policies below.

3.1 Sources of Information

We can classify task distribution strategies based on the sources of information used to make distribution decisions. Table 2, an extension of the classification scheme of Shu and Kalé [31], summarizes the six categories we used. These classifications do not incorporate the possibilities for selected destinations. For example, they cannot differentiate a random placement strategy that

Strategy Type	Strategy Description
Type I	Use no information (e.g., a deterministic or probabilistic static scheduling algorithm).
Type II	Use only the local load, or information about the current node.
Type III	Calculate the status information by collecting *load* information from neighbors.
Type IV	Calculate the status information by collecting *status* information from neighbors.
Type V	Calculate the status information by collecting *load* information from all nodes in the system.
Type VI	Calculate the status information by collecting *status* information from all nodes in the system.

Table 2: Classification of Load Distribution Strategies

places processes on neighboring nodes from one that places them on arbitrary nodes; both are treated as type I strategies.

3.2 Load and Status Metrics

Some strategies only use load information; others use a combination of load and status information. In general, we used the total number of tasks assigned to a node for the load metric. The status metric selected varies with the distribution strategy. We implemented the gradient metric of Lin [25] and two metrics that infer load over a broader region. We call these metrics *load of peers* (**LP**) and *status of peers* (**SP**); both use load and status information from a *peer group*, or topologic neighborhood. Let $T_{i,j}$ be the communication delay from node j to node i. This value is measured for each message transmission; we assume the value $T_{i,j}$ is recorded by the communication processor. Similar, $L_{i,j}$ is the load of node j as known by node i, and $S_{i,j}$ is the status of node j as known by node i. We define **LP** and **SP** as follows, where w_i is a weighting of local and non-local information, and P is the set of node peers.

- **Load of Peers (LP):** This metric uses the last known load values of the local node and the node's peers:

$$S_{i,i} = w_i L_{i,i} + (1 - w_i) \frac{\sum_{p \in P} T_{i,p} L_{i,p}}{\sum_{p \in P} T_{i,p}}$$

- **Status of Peers (SP):** This metric uses the last known load value of the local node and the status values of the node's peers:

$$S_{i,i} = w_i L_{i,i} + (1 - w_i) \frac{\sum_{p \in P} T_{i,p} S_{i,p}}{\sum_{p \in P} T_{i,p}}$$

Both **LP** and **SP** incorporate local information, represented by the first term, and non-local information from the peer group in the second term. The contribution of each peer to the non-local information is weighted by the communication time to that node, and the sum is normalized by the total communication time of the peer group. The motivation behind these metrics, like Lin's

gradient metric, is that they may be better able to place new tasks in a lightly loaded area of the network rather than simply the least loaded node in the network. In general, all of the status-based strategies (**LP**, **SP** and gradient scheduling) had poorer performance than the simpler strategies we describe here; [13] provides a complete comparison.

3.3 Information Dissemination

We assume that load and status information are piggybacked on normal messages, providing passive information dissemination. Several strategies disseminate information *actively* using explicit load messages. Load messages have lower transmission priority to reduce interference with program communication, and they are generated by the communication processor, eliminating computation overhead. We implemented three active methods. The first, *active adjacent* updating, dispatches messages to neighboring nodes whenever the current load or status value changes. The second, *active radius* does the same for a larger number of nodes delimited by a *region radius*. The last method, *active broadcast*, broadcasts status information at fixed time intervals of at most one hundred microseconds; no broadcast is done if the load has not changed since the previous broadcast. The last distribution method relies on *true information*, or completely accurate load information. In the simulated system, this involves reading load values directly; in an actual network, this would be implemented using a global backplane interconnection [11].

3.4 Distribution Policies

We implemented five policy families:

- **Random:**

  This policy randomly selects a distribution destination from a set of candidates. In global random placement, *all* nodes in the system are candidates. In local random placement, only the node itself and topologically adjacent nodes are candidates. In *Global Random Walk*, a node is selected as in global random placement. However, the selected node can elect to forward the task if it belives other nodes are less busy. This process continues until a forwarding limit is reached or a node believes it is the least loaded node in the network.

- **Gradient:**

  The gradient policy implements Lin's gradient scheduling algorithm. Each node is classified as having an idle, abundant, or neutral state, according to the algorithm of [25]. If a task is distributed, it can only be sent to an adjacent node; that node may elect to continue transferring the task.

- **Min:**

  This policy places a new task on the candidate node with the minimum value of the load metric. Once placed, tasks cannot be transferred to other nodes.

- **Drift:**

  This policy is similar to **Min**; however, invocation requests can be forwarded after they arrive at a node. The NCWN algorithm, an implementation of the Naive Contracting Within a Neighborhood strategy [19, 31], is a member of the **Drift** policy that uses the load values of adjacent nodes to select locations for new tasks.

Strategy Name	Policy	Class	Peer Radius	Status Value	Update Interval
LT	**Min**	Type V	2	True Information	N/A †
GT	**Min**	Type V	Global	True Information	N/A
LR	**Random**	Type I	1	N/A	N/A
GR	**Random**	Type I	Global	N/A	N/A
GRWalk	**Random Drift**	Type V	Global	N/A	N/A
NCWN	**Drift**	Type III	1	N/A	Passive
NCWN.A	**Drift**	Type III	1	N/A	Active Adjacent
GRAD	**Gradient**	Type III	1	Gradient	Passive
GRAD.A	**Gradient**	Type III	1	Gradient	Active Adjacent
DLL	**Drift**	Type III	2	N/A	Passive
DGL	**Drift**	Type V	Global	N/A	Passive
DGL.A	**Drift**	Type V	Global	N/A	Active Radius
DGL.B	**Drift**	Type V	Global	N/A	Active Broadcast
GRD	**Random& Min**	Type V	Global	2	Passive
GRD.A	**Random& Min**	Type V	Global	2	Active Adjacent
GRD.T	**Random& Min**	Type V	Global	2	True Information

† Not Applicable

Table 3: Subset of Distribution Strategies Examined

- **Region:**

  This strategy is similar to **Drift**, but differs from all of the previous methods by explicitly using load *and* status information. The first time an invocation request is transferred, it is sent to the candidate node with minimum *status* value; on subsequent transfers, it is sent to the candidate node with the minimum *load* value.

In [13, 26], we examined several possible task placement strategies (i.e., instances of each policy family with different sources of load and status information). Table 3 summarizes the parameters used in a subset of the strategies examined in [13, 26]. Many strategies are omitted because of space considerations, and because they were uniformly ineffective. The bulk of the strategies have names denoting their policy family, placement class, status metric (if applicable) and status dissemination method. The *peer radius* defines defines the maximum distance from the originating node where a strategy can place a new task. For example, *DLL* is a member of the **Drift** family, it only uses local load information from nodes within two hops, and it can place new tasks on nodes within two hops; because it is a member of the **Drift** family, task invocation requests may be forwarded by the selected node before the task is finally placed. Likewise, the *DLL.A* strategy is akin to *DLL*, but uses *active adjacent* information distribution.

4 Load Distribution Workloads

Given a simulated multicomputer that implements different load distribution strategies, we needed workloads, or example programs, to evaluate the distribution strategies. There are few extant multicomputer environments that support dynamic task invocation. This complicates load distribution studies, because there are few task creation traces that represent the execution of actual programs. Hence, we used both *captured* and *synthesized* workloads. The Chare Kernel [20] is a portable environment for distributed computation using *chares*, or small, very lightweight task with restricted control flow. The total number of tasks and the execution time of each task are unknown at compile time. We used four captured workloads from Chare Kernel programs; two were C programs and two were compiled ROPM Prolog [27] programs. The externally observable behavior of each Chare Kernel task was recorded in a timestamped trace. Computation time was measured using a microsecond timer and scaled to simulate a processor executing approximately ten million instructions per second. A similar process was used to trace the execution of a multigrid algorithm implemented using *Concurrent Aggregates* (CA) [7], a concurrent programming language developed at MIT.

Although Concurrent Aggregates and the Chare Kernel implement different computing environments, both use a similar set of abstractions, such as task creation and message transmission. The programs were traced by recording the occurrence of the common abstractions, allowing us to gather trace data from either environment.

Below, we briefly describe each of the measured workloads.

Queens: Solves the ten-queens problem for a chess board ten squares on each edge using a divide-and-conquer algorithm. Communications occur only between a task and its children.

Cubes: Searches for a number N that can be expressed as the sum of two cubes in two different ways (i.e., $N = A^3 + B^3 = C^3 + D^3$). As in the **Queens** program, the communications occur only between a task and its children.

	Minmax	IDA*-15	Queens	Cubes	MultiGrid
Environment	CK	CK	CK	CK	CA
Number of Tasks	2,801	5,676	3,874	1,933	949
Task Depth	4	23	15	3	17
Total Messages	2,800	5,646	8,189	5,313	28,843
Total Message Size (bytes)	11,200	90,336	393,072	255,024	772,430
Execution Time (microseconds):					
Infinite	2,867	885	459,001	1,512,360	152,022
Sequential	247,286	57,400	1,276,860	6,077,070	4,804,660
Max. Speedup	86.23	64.83	2.78	4.02	31.6
Task Computation Time (microseconds):					
Mean	88.3	10.1	329.6	3,143.9	5,062.9
Std. Dev.	105.72	0.26	274.83	8,436.68	3,639.29
Min	47.10	10.00	130.80	172.60	86.30
Max	2,327.10	27.30	7,987.20	369,690.70	20,112.7
Messages per Task:					
Mean	1.00	0.99	2.11	2.75	30.39
Std. Dev.	0.02	0.07	1.13	2.80	22.03
Min	0.00	0.00	0.00	0.00	0.00
Max	1.00	1.00	7.00	31.00	102.00
Message Size per Task (bytes):					
Mean	4.00	15.91	101.46	131.93	813.94
Std. Dev.	0.08	1.16	54.01	134.31	557.26
Min	0.00	0.00	0.00	0.00	0.00
Max	4.00	16.00	336.00	1488.00	2,375.00

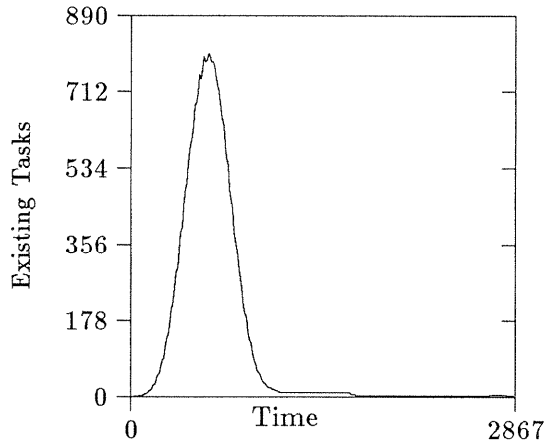
Table 4: Measured Kernel Workload Characteristics

Minmax: Plays the game of Othello on a board of eight squares on each edge by competing against itself. As in the above programs, communications occur only between a task and its children, and furthermore, these communications occur only on creation and termination of the children.

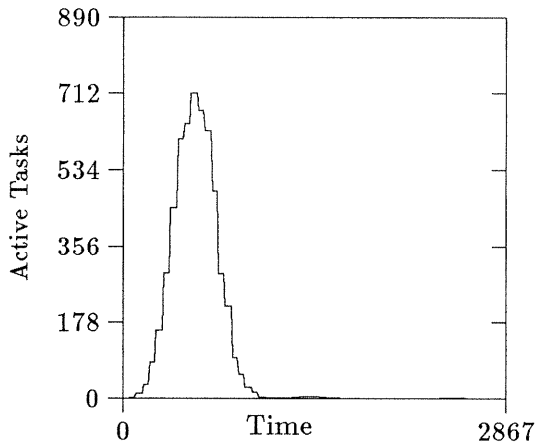
IDA*-15: Attempts to solve a randomly generated 15-puzzle using an iterative, deepening A* search. As in **Minmax**, communications occur only between a task and its children on creation and termination. The only exception is that execution is terminated when the solution to the puzzle is found.

MultiGrid: Solves a finite difference equation using the multigrid algorithm. Unlike most of the previous Chare Kernel programs, each task of this Concurrent Aggregates program sends and receives multiple messages.

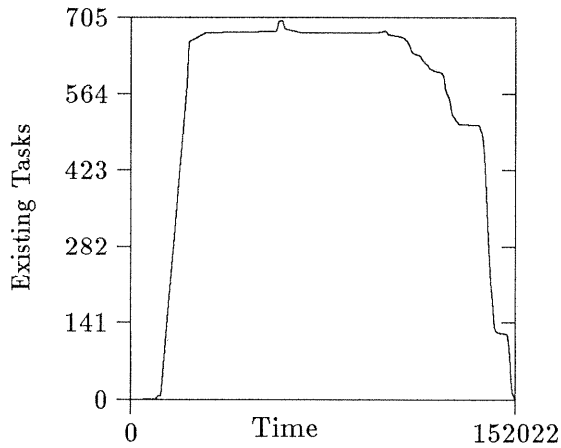
The synthesized workloads were generated from abstract descriptions of program behavior. These workloads provide a range of task behavior absent in the captured workloads. Taken together, the range of task activities (total computation time, total number of messages sent and average



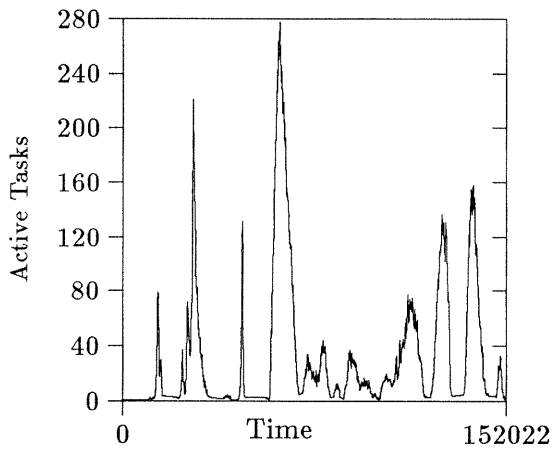
(a) Task Index for `Minmax`



(b) Concurrency Index for `Minmax`



(c) Task Index for `MultiGrid`



(d) Concurrency Index for `MultiGrid`

Figure 2: Task and Concurrency Indices

	Compute-1	Compute-2	Message-1	Message-2
Number of Tasks	3,345	1,252	103	328
Task Depth	1	3	3	3
Total Messages	3,344	1,324	6,115	52,840
Total Message Size (bytes)	84,942	68,176	766,588	6,780,636
Execution Time (microseconds):				
Infinite	516,590	759,622	52,211	108,780
Sequential	17,415,600	25,549,000	315,645	2,375,960
Max. Speedup	33.71	33.63	6.05	21.84
Task Computation Time (microseconds):				
Mean	5,206.45	20,406.51	3,064.52	7,243.77
Std. Dev.	9,962.46	37,598.05	4,730.35	10,759.03
Min	1.85	8.27	2.12	4.23
Max	500,000.00	500,000.00	33,804.74	88,276.90
Messages per Task:				
Mean	1.00	1.06	59.37	161.10
Std. Dev.	0.02	0.26	113.80	286.33
Min	0.00	0.00	1.00	1.00
Max	1.00	3.00	984.00	2,391.00
Message Size per Task (bytes):				
Mean	25.39	54.45	7,442.60	20,672.67
Std. Dev.	24.20	53.33	14,080.55	36,997.24
Min	0.00	0.00	20.00	26.00
Max	200.00	390.00	120,582.00	303,381.00

Table 5: Synthetic Workload Characteristics

message size) spans two orders of magnitude. The salient characteristics of the workloads are summarized in Tables 4–5.

For each workload, we measured both the single processor execution time and the minimum potential execution time for each program. The latter, obtained by eliminating all processor contention and communication delays, corresponds to the delay on the longest possible path through the program flowgraph. We use this to compute the maximum possible speedup for each program. Tables 4–5 also show metrics for the total computation and communication activity for each program; these provide a static description of program activity.

The dynamic activity of a program is more difficult to summarize. Figure 2 shows the dynamic characteristics of two representative programs from our set of test workloads. Figures 2(a) and 2(c) show the *total number* of extant tasks, whereas Figures 2(b) and 2(d) show the number of *active* tasks, or tasks actually able to execute. In both cases, we assume no processor contention and no communication delay. Differences are due solely to the intertask data dependencies caused by message exchange. Although the envelope of these figures would change with different load placement strategies or scheduling algorithms, the total area under each curve is fixed — it represents the total computation time of all tasks.

Although concurrency profiles clearly differ across programs, they do not completely characterize program dynamics. The pattern of task interactions and the structure of the task creation tree. both influence the potential performance. In the `Minmax` program, most tasks are created shortly after program begins, and the task ancestry relation forms a balanced tree. most of the tasks of the `MultiGrid` program also are created near the beginning of execution. However, the task creation graph for `MultiGrid` is skewed; two tasks are the parents of over half of all the other tasks.

5 Comparison of Distribution Strategies

The large number of distribution strategies, coupled with nine program traces, provides an unwieldy array of data, far too great for succinct analysis. Based on our data analysis, we have concluded that no single task placement strategy is unambiguously best across all workloads; the spectrum of possible behaviors is simply too broad. Thus, we have distilled the data to infer general trends and highlights.

Although static performance metrics (e.g., speedup) demonstrate the performance of different load placement strategies, they do not illuminate the *reasons* for that performance. We have found that dynamic, or time varying, metrics provide that insight; in §6, we will examine a subset of the workloads in more detail, using dynamic information to explain the performance of different strategies. First, however, we present an overview, showing summary statistics for several simulation studies, and offer a few observations based on our data analysis.

The strategies of Table 3 differ in the number of times a task invocation request can be *transferred.* All our task placement strategies are *server initiated* [9] (i.e., it is the responsibility of overly busy nodes to disperse tasks); however, the node selected during task distribution can delegate the new task by itself transferring the task to another node. If tasks are short lived (e.g., as in the `Minmax` program), repeatedly transferring task invocation requests can have a deleterious effect on the speedup, as Table 6 shows. Simply put, the overhead for multiple transfers exceeds the cost of quickly placing and executing the short lived tasks. The *DLL* strategy transfers tasks to the least loaded adjacent node, but allows tasks to be incrementally transferred across the diameter of the network. The *DGL* strategy is similar, but can select *any* node in the network, not simply adjacent nodes. Unless aggregate processor utilization is extremely low, the local strategy, *DLL*, performs slightly better than the global strategy, *DGL*. Why? Initially, we attributed this to enhanced communication locality; however, comparison with the global random strategy (*GR*) suggests otherwise. The *GR* strategy places tasks on randomly selected nodes drawn from the entire network, and, in general, it performs almost as well as the *DLL* strategy. We concluded that *DGL*, confused by *stale information*, repeatedly forwards tasks in pursuit of the chimera of idle nodes; see Figures 5 and 12.

Observation 1 *Stale information hampers global task distribution strategies that use load information (i.e., Type II – Type VI strategies).*

The *DGL* strategy attempts to place tasks on nodes with minimum load, but it does not have accurate load information; thus, tasks are transferred excessively. This is a fault of point-to-point networks; intermediate nodes learn nothing from messages transiting a node. The *DLL* strategy also suffers from stale information, but to a lesser extent; it considers only neighboring nodes, and tasks are more likely to become ensnared in local "load minima," reducing the likelihood of

	Nodes In System							
Strategy	2	4	8	16	32	64	128	256
Unlimited Process Forwarding								
DLL	1.99	3.95	7.71	14.64	22.34	35.39	53.92	64.16
DGL	1.99	3.95	7.74	13.35	20.51	29.61	45.68	70.19
GRWalk	1.98	3.93	7.69	14.55	18.57	25.48	39.13	67.17
Limited Process Forwarding								
NCWN	1.98	3.95	7.73	14.86	26.91	43.93	61.68	70.17
MLL	1.99	3.94	7.76	14.73	26.09	38.73	49.02	50.89
MGL	1.99	3.94	7.76	14.78	25.57	33.85	51.25	59.53
D2L.L	1.99	3.95	7.75	14.79	27.01	42.64	54.40	60.66
D2G.L	1.99	3.95	7.73	14.86	25.86	30.22	51.48	68.31
GR	1.96	3.77	7.04	12.63	21.85	35.39	49.18	62.36
Forwarding With Information Dissemination								
NCWN.A	1.99	3.95	7.77	14.97	27.57	44.83	65.18	70.10
DLL.A	1.99	3.95	7.74	14.45	21.22	32.65	52.20	68.89
DGL.A	1.99	3.95	7.71	13.24	17.40	26.94	42.20	68.59

(a) `Minmax` Speedup

	Nodes In System							
Strategy	2	4	8	16	32	64	128	256
Unlimited Process Forwarding								
DLL	1.18	2.95	6.28	10.05	15.43	19.48	22.40	23.37
DGL	1.18	2.87	6.23	9.84	14.19	18.99	22.53	23.84
GRWalk	1.86	2.84	5.10	9.22	14.03	19.01	22.32	23.94
Limited Process Forwarding								
NCWN	1.93	2.12	3.96	5.25	6.98	9.48	10.32	11.17
MLL	1.18	1.93	2.42	3.66	4.45	4.75	4.88	5.43
MGL	1.18	1.96	2.68	4.84	6.10	8.34	17.71	23.49
D2L.L	1.93	2.98	4.73	9.25	13.64	16.75	19.07	19.98
D2G.L	1.93	2.96	5.70	9.96	11.54	18.75	21.90	23.96
GR	1.92	3.48	6.05	9.86	14.70	18.88	22.24	23.62
Forwarding With Information Dissemination								
NCWN.A	1.93	2.79	2.68	4.61	5.92	7.42	8.36	8.18
DLL.A	1.19	2.83	6.21	9.95	15.06	19.41	22.01	23.95
DGL.A	1.18	2.78	6.27	9.92	14.42	18.78	22.58	23.83

(b) `MultiGrid` Speedup

Table 6: Deleterious Effect of Process Forwarding

	Nodes In System							
Strategy	2	4	8	16	32	64	128	256
No Information Dissemination								
GR	1.96	3.77	7.04	12.63	21.85	35.39	49.18	62.36
LR	1.95	3.62	5.72	8.44	10.52	14.19	17.88	19.93
Passively Disseminated Information								
GRD	1.99	3.94	7.77	14.84	27.16	44.47	52.55	61.44
DLL	1.99	3.95	7.71	14.64	22.34	35.39	53.92	64.16
NCWN	1.98	3.95	7.73	14.86	26.91	43.93	61.68	70.17
Actively Disseminated Information								
GRD.A	1.99	3.95	7.77	14.77	27.42	44.40	55.84	63.58
DLL.A	1.99	3.95	7.74	14.45	21.22	32.65	52.20	68.89
NCWN.A	1.99	3.95	7.77	14.97	27.57	44.83	65.18	70.10
Accurate Information								
GT	1.99	3.95	7.76	15.00	27.99	44.05	44.33	58.96
LT	1.99	3.95	7.77	14.89	27.36	47.51	68.44	74.33
GRD.T	1.99	3.96	7.74	14.92	27.82	47.26	68.37	79.41

(a) `Minmax` Speedup

	Nodes In System							
Strategy	2	4	8	16	32	64	128	256
No Information Dissemination								
GR	1.92	3.48	6.05	9.86	14.70	18.88	22.24	23.62
LR	1.91	2.84	3.75	4.63	5.28	6.14	6.87	7.56
Passively Disseminated Information								
GRD	1.91	3.38	6.21	10.13	14.32	18.59	20.97	23.56
DLL	1.18	2.95	6.28	10.05	15.43	19.48	22.40	23.37
NCWN	1.93	2.12	3.96	5.25	6.98	9.48	10.32	11.17
Actively Disseminated Information								
GRD.A	1.92	3.42	6.17	10.35	14.86	19.00	21.93	23.95
DLL.A	1.19	2.83	6.21	9.95	15.06	19.41	22.01	23.95
NCWN.A	1.93	2.79	2.68	4.61	5.92	7.42	8.36	8.18
Accurate Information								
GT	1.19	1.97	2.98	2.81	5.36	7.68	9.62	15.50
LT	1.19	2.00	2.98	2.95	5.54	6.91	9.04	14.59
GRD.T	1.89	3.49	6.30	10.57	16.11	20.91	23.55	24.46

(b) `MultiGrid` Speedup

Table 7: Distribution Strategy Comparison

forwarding. For the `Minmax` program, both *DLL* and *DGL* spent too much time *balancing* the load; comparison with `MultiGrid` shows that this is a function of task lifetime. The average task lifetime in `MultiGrid` was 57 times longer than in `Minmax`, and the extra forwarding time was mitigated by the better load distribution. Clearly, the time constant for information propagation must be commensurate with the time between application program task state changes.

Observation 2 *Load information staleness is correlated with task lifetime.*

To understand the interaction of information staleness and task forwarding, we modified the *DLL* and *DGL* strategies to forward tasks no more than twice; these strategies are labeled *D2L.L* and *D2G.L* in Table 6. These modified strategies are similar to the *NCWN* strategy, except tasks are not *forced* to be forwarded at least once, as in *NCWN*. For the `Minmax` program, with its short lived tasks, limiting the number of possible task transfers often increases the speedup; short lived tasks can begin executing quickly, and the placement decisions are no worse than before — most transfers were based on inaccurate load information. In contrast, the long lived tasks of the `MultiGrid` program benefit from repeated transfers; the cost of finding idle nodes, though substantial, is repaid by better load balance.

In the expectation that increasing the frequency of load information distribution would allow the *DLL* and *DGL* strategies to make more opportune placement decisions, we also modified the *DLL* and *DGL* strategies to actively distribute load information to adjacent nodes, yielding the *DLL.A* and *DGL.A* strategies of Table 6. Surprisingly, unless the mean processor utilization was very low (i.e., large systems), this *reduced* program speedup for the `Minmax` program. Why? Using the *DLL* strategy, we conjecture that tasks were trapped in "local minima," whereas in the *DLL.A* strategy, more frequent information dissemination increased task forwarding. Thus, `Minmax` tasks spent more time being forwarded. However, for the long lived tasks of the `MultiGrid` program, there is little performance change with active distribution of load information.

Observation 3 *Distribution strategies should limit task forwarding. Ideally, the forwarding limit should adapt to system communication overhead and expected task computation time.*

The *GT* and *LT* strategies use *accurate* or *true information* when placing tasks; they rely on load information obtained from our simulation system. As such, they are not implementable, but they do provide a point of reference. Table 7 compares these accurate methods to the *informationless* strategies *LR* and *GR*. These informationless strategies bound the effort that should be expended collecting accurate status information. In general, the informationless strategies perform quite well.

With accurate information, both *LT* and *GT* work well in general, but exhibit anomalous degradations. Initially, the reason for this was unclear, but was explicated by an analysis of the temporal performance data discussed in the next section. We assumed that the *LT* and *GT* methods could be implemented with a limited number of out-of-band broadcast messages [11]. When a node selects a destination, it enqueues a task invocation message for transmission. However, tasks are often created in bursts, generating many messages with significant message transmission delays. The selected node continues to advertise a low load while the invocation messages are transmitted, and it is repeatedly perceived as lightly loaded although there are numerous tasks enroute. This particularly affects the *GT* strategy, because *all* nodes may select a single node repeatedly.

The *LT* strategy is less affected by tasks enroute, because each node considers only a subset of the total nodes. The *LT* strategy performs better than *LR* because local strategies are very sensitive

to poor placement due to *shallow task instantiation trees*. Most tasks are created by only a few parents. If one views the history of task creations as a tree, called a *task instantiation tree*, `Minmax` has a shallow, broad tree. The parent process in `Minmax` creates 2800 tasks in a tree of four levels, with seven children for each non-leaf task. By comparison, `MultiGrid` has a much deeper tree, but most tasks are directly created by two parent tasks. The remaining tasks contribute greatly to the tree depth, but perform little computation; thus, the computationally intensive tasks are created in a shallow fashion, similar to `Minmax`. Methods that distribute tasks only to neighboring nodes are unable to disperse a large number of tasks over many nodes; tasks simply congregate near their point of origin, leaving most network nodes idle. The *LT* strategy effectively pushes child tasks as far from the parents as possible; the *LR* strategy selects nodes randomly, returning some children to the nodes with parent tasks.

Observation 4 *Local placement strategies are inappropriate for programs with a shallow task instantiation tree.*

We have seen that selecting a node globally is beneficial, because local methods cannot disperse large numbers of long lived tasks. Moreover, it is difficult to maintain accurate global information. Thus, randomly selecting a global node (i.e., the *GR* strategy) is appealing. Intuitively, however, one can improve the *GR* strategy by combining the value of accurate local information with the dispersion of global random placement. To test this hypothesis, we implemented the *GRD* strategy ("global random drift"). Each task can be forwarded twice; initially, a node is selected randomly and globally, with a single, subsequent forwarding step to an adjacent node. Table 7 shows that the *GRD* strategy is very successful; analysis of other workloads confirms this.

Acquiring load information has benefits and associated costs. In §3.3 we mentioned several information dissemination techniques, each with an associated cost. Passive dissemination is amenable to implementation on systems without message processors; without a message processor, active dissemination consumes substantial computation processor time. True dissemination, as used by *LT* and *GT*, requires special hardware; of these, *LT* is easier to implement.

To compare the benefits of these dissemination methods, we implemented three variants of several strategies using passive, active or accurate information, and compared them to strategies that use no information. We chiefly considered strategies that use local information; Table 7 summarizes the measured speedups. In general, the speedups do not justify using message processors solely for load information dissemination, although there are numerous other benefits from their use. The improvement for the *GRD.T* strategy indicates the benefit of local true information is slight; it may justify modified network hardware, but only if other uses for those modifications exist.

How realistic are these results? Users of first-generation hypercube architectures, such as the Intel iPSC/1 or the Ametek System/14, would typically recoil at the idea of the *GRD* strategy. For those architectures, communication locality dictates more limited task distribution. With second generation architectures, such as the Intel iPSC/2 and Symult 2010, locality has become relatively unimportant; however, high communication latency and cut-through networks increase the importance of the *number* of communication events.

Our simulated network posited the low communication latency possible from a research architecture; what is the effect of the higher communication latency on our results? A simplified model of communication delay [16] in wormhole and circuit-switched networks includes a per-message software overhead, a per-hop transit delay for messages switched through a node and a per-byte delay dictated by network bandwidth. When sending a fixed size message, the software overhead

typically dominates the transit delay in a lightly loaded network. As we will see in §6.3, increased software overhead decreases the performance of all placement strategies.

To summarize, our simulation studies suggest that, in general, random, global strategies perform relatively *better* than strategies that rely on forwarding tasks (e.g., *DLL*) or more accurate local information (e.g., *NCWN*) for most workloads. The `Message-2` workload is an exception; it is very communication intensive and very long running. Poor placement decisions delay message generation, and thus message reception, making performance very sensitive to the initial placement. Without *a priori* information concerning the application, we believe that simple placement strategies such as *GR* perform well; however, they must be augmented with *load redistribution* mechanisms to compensate for poor initial placements when programs contain long running tasks.

6 Detailed Comparisons

To provide more insight into the dynamic behavior of each task placement strategy, we gathered a variety of time dependent information about the aggregate system state, the *spatial* and *temporal* distribution of tasks, and the observed error in load estimates. Each is described briefly below, followed by case studies of the `Minmax` and `MultiGrid` programs. Where appropriate, reported data are shown with ninety percent confidence intervals, obtained via three simulated program executions, each with differing seeds for the random number generators.[1]

6.1 System Metrics

We recorded several metrics in addition to the traditional *speedup* measure, including CPU and I/O processor utilization, task placement cost, processor and message waiting time, and message transmission time. The first of these, CPU and I/O processor utilization, reflects use of the computation and communication processor, respectively. Intuitively, one would expect the highest average CPU utilization from a placement strategy that most equitably distributed tasks across the nodes; we shall see, however, that this is not always the case. The average I/O processor utilization is one measure of the communication network load; it includes both application program task interactions and, for strategies that actively disseminate load information, the load information messages. Although, strictly speaking, I/O processor utilization ignores the utilization of the actual circuits or wires used, other studies [14, 15] have shown that the I/O processor saturates before the network transport medium.

Following its creation, a task is quiescent during the search for a node that satisfies the task placement strategy's criteria. If the task lies on the parallel application program's critical path, this idle time is a direct contributor to total program execution time. To quantify the expected cost for task placement, we recorded both the number of times a task invocation request is transferred and the time needed to identify a recipient node.

Although the goal of load distribution is maximizing effective parallelism, not all the tasks of a program can be concurrently active. If several tasks must share a resource, be it a CPU, communication processor or the network, resource contention is inevitable, and some tasks must wait. Moreover, task graph precedence constraints also force some tasks to wait for messages from other tasks. Both long task placement delays and resource contention after placement exacerbate

[1] Temporal data are chosen from representative executions. Although the data in this section may appear to differ from that in §5, both agree within the precision of the confidence intervals.

these message waiting delays. Thus, we measured both the mean time a task spent waiting for a processor and the mean time spent waiting for messages; these metrics show where tasks spend the majority of their time.

To study communication activities, we measured the time to send messages between nodes. The internode message transmission time depends on the condition of the network and the number of messages already in the transmission queue.

6.2 Spatial and Temporal Metrics

Our system metrics encompass *all* nodes over the *entire* application program's execution time; however, variability exists in both the spatial and temporal domains. To understand the causes of this variability and its effects on high-level performance measures (e.g., speedup), we recorded a small set of spatio-temporal data, showing performance data over time for individual nodes, and temporal data, showing the data over time for all nodes. In each case, the data was recorded at a series of regular time intervals and encompasses the events within that time interval.

Currently, the temporal data include the computation processor utilization, the number of tasks per node, the number of transfers when placing a task and the *average load estimation error.* The latter is the difference between the *estimated* load and the *actual* load. If $L_{i,j}$ is the load of node j as known by node i, than the average load estimation error at a point in time t for N nodes is

$$\hat{L}(t) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} (L_{i,j} - L_{j,j})}{N^2},$$

if we assume that $L_{j,j}$ (the load of j as known by j) is the actual load on node j. Note that a negative value indicates that node i is *underestimating* the load of node j, whereas positive values indicate it is *overestimating* the load.

6.3 Case Study: `Minmax`

Of our test workloads, the `Minmax` program has the greatest potential concurrency, and its mean task computation time is slightly greater than its average message transmission time, given our hardware assumptions. Figure 2(b) shows the potential concurrency in `Minmax`, assuming an infinite number of processors; most tasks are created shortly after program execution begins. However, no single task creates many child tasks; instead, the task ancestry relation forms a shallow, balanced tree. Nevertheless, the initial burst of tasks tests the ability of any placement strategy to quickly and effectively distribute tasks.

For a detailed analysis of task placement strategies, we selected the local and global variants of the **Random**, **Min**, and **Drift** policies. Recall that *LR*, *MLL*, and *DLL* are *local* strategies that distribute tasks only to neighboring nodes, whereas *GR*, *MGL*, and *DGL* are *global* strategies that can distribute tasks to any node in the system. As we have seen, the *LR* strategy considers only topologically adjacent nodes, whereas the *MLL* and *DLL* strategies have a larger *peer radius* and consider all nodes within two hops.[2]

Figure 3 shows the observed speedups for the `Minmax` program. We include a hypothetical, *ideal* speedup for reference. This ideal speedup is equal to the number of nodes but is bounded by the

[2]The properties of the hypercube topology imply that the number of neighbors within a fixed distance increases with the system size.

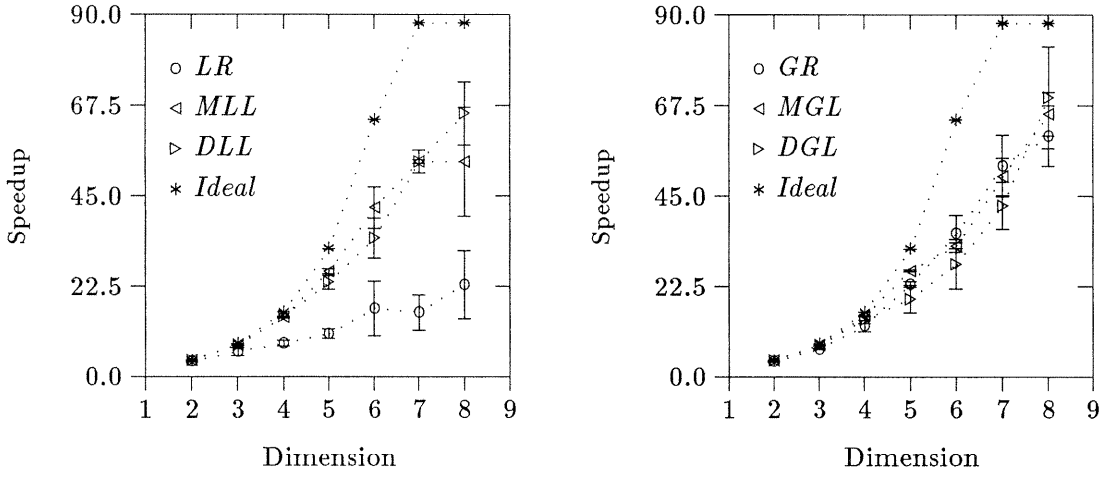


Figure 3: `Minmax` Workload: Speedup

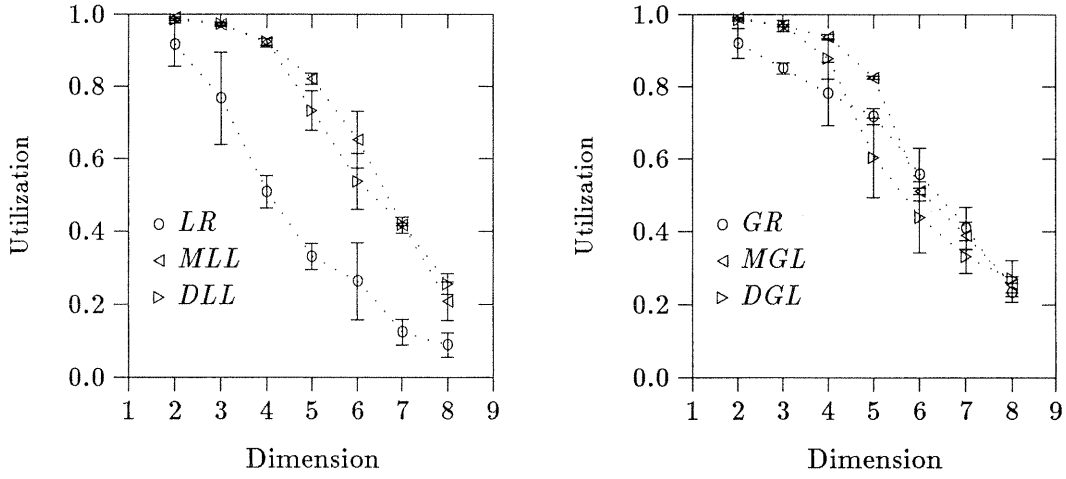


Figure 4: `Minmax` Workload: Processor Utilization

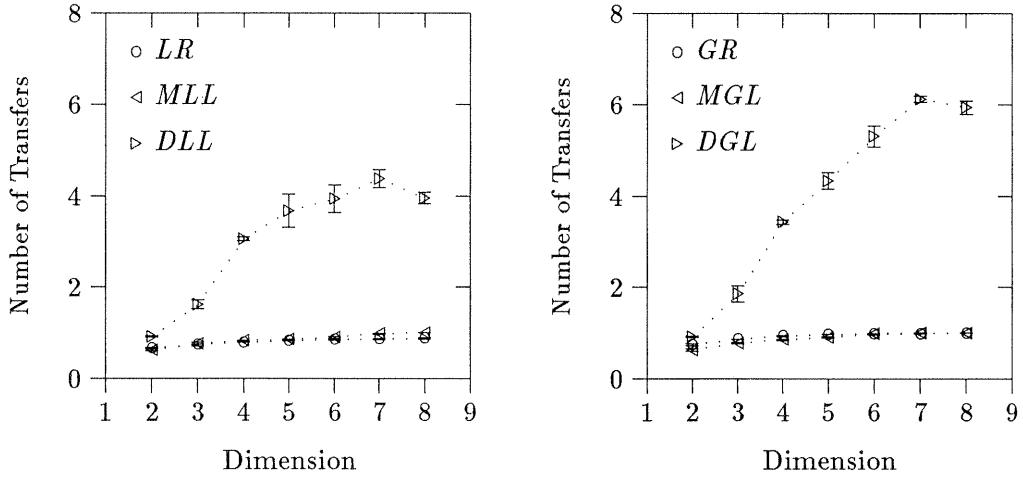


Figure 5: `Minmax` Workload: Number of Transfers Per Placement

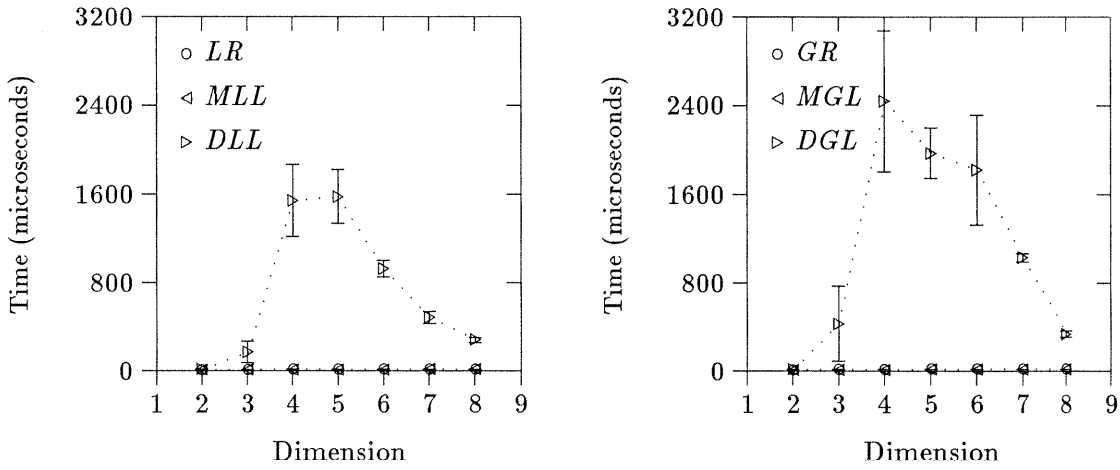


Figure 6: `Minmax` Workload: Time Spent Finding a Destination Node

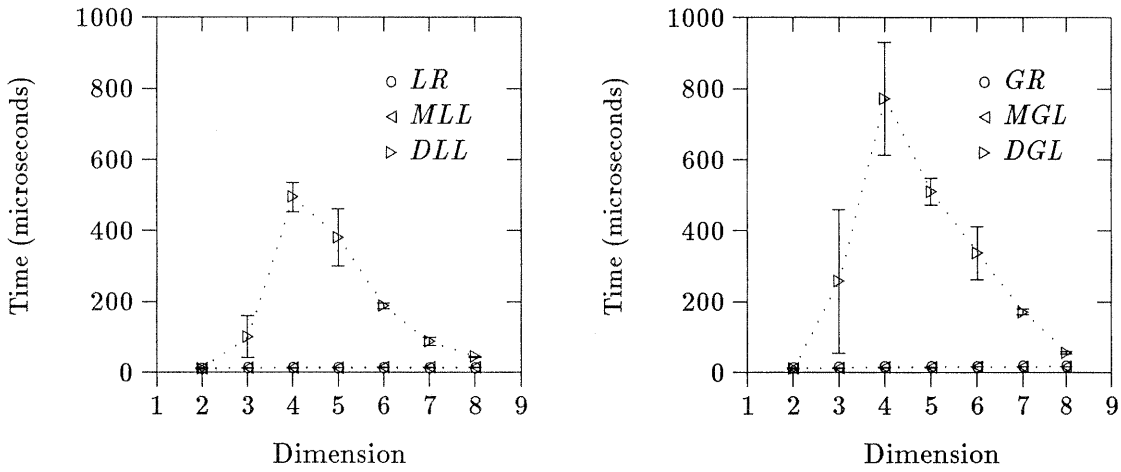


Figure 7: `Minmax` Workload: Average Message Transmission Time

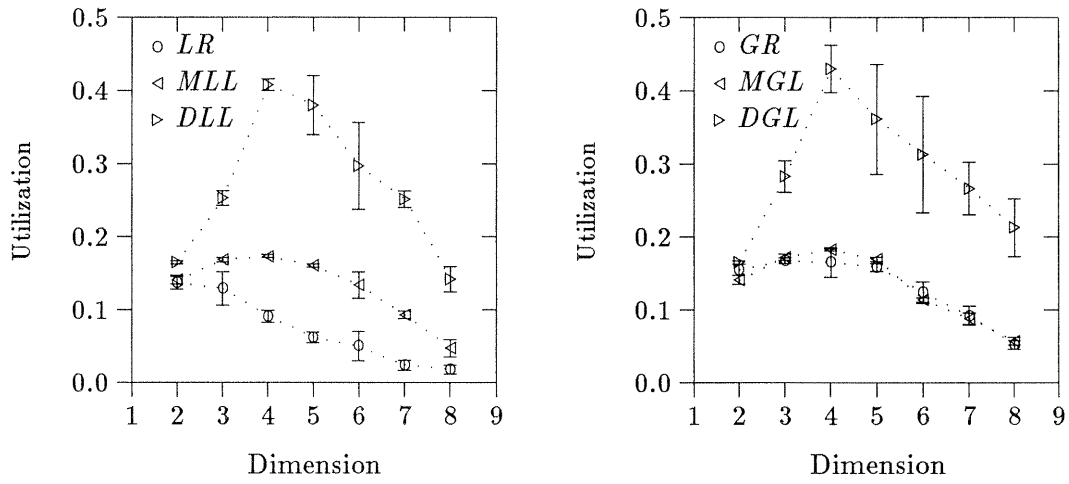


Figure 8: `Minmax` Workload: I/O Processor Utilization

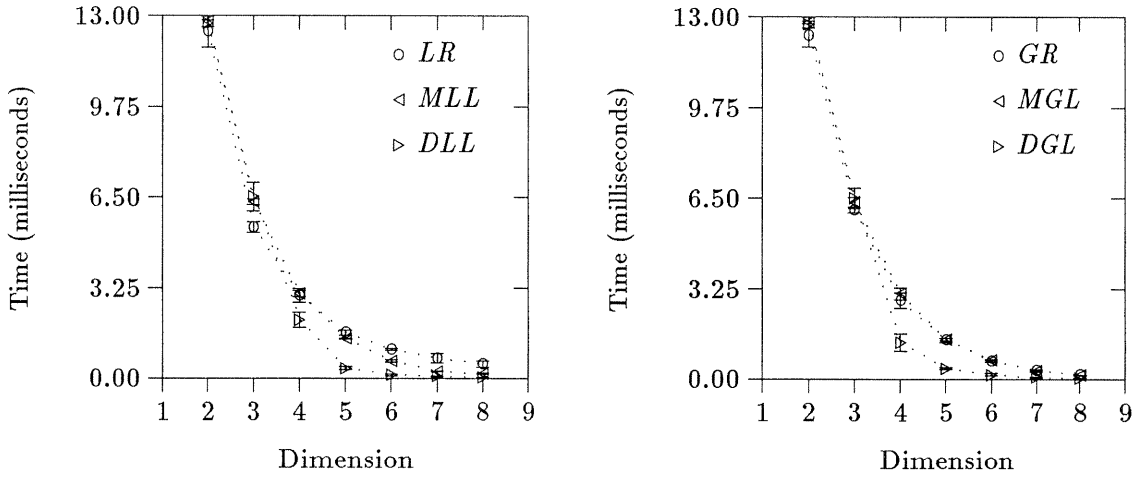


Figure 9: `Minmax` Workload: Time Spent Waiting for a Processor

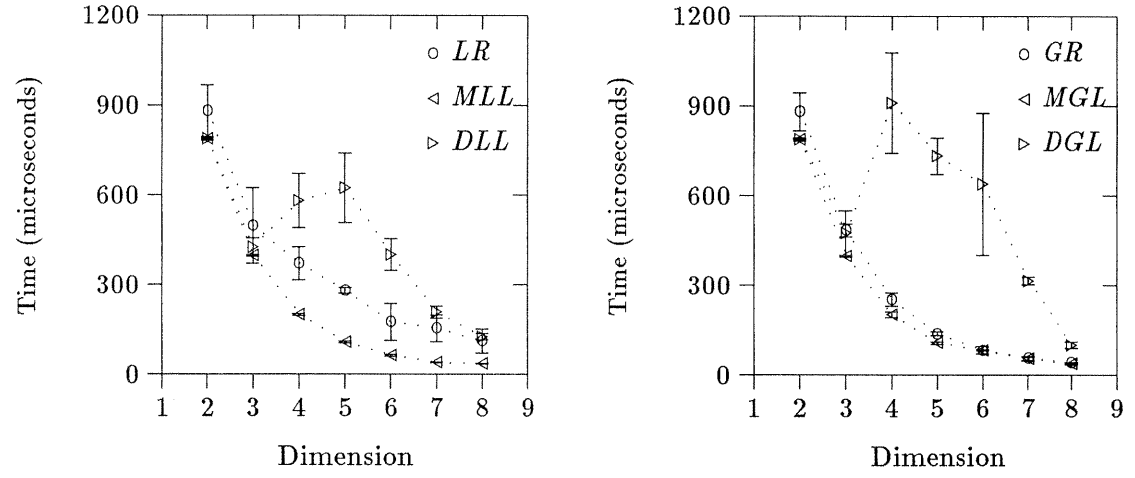
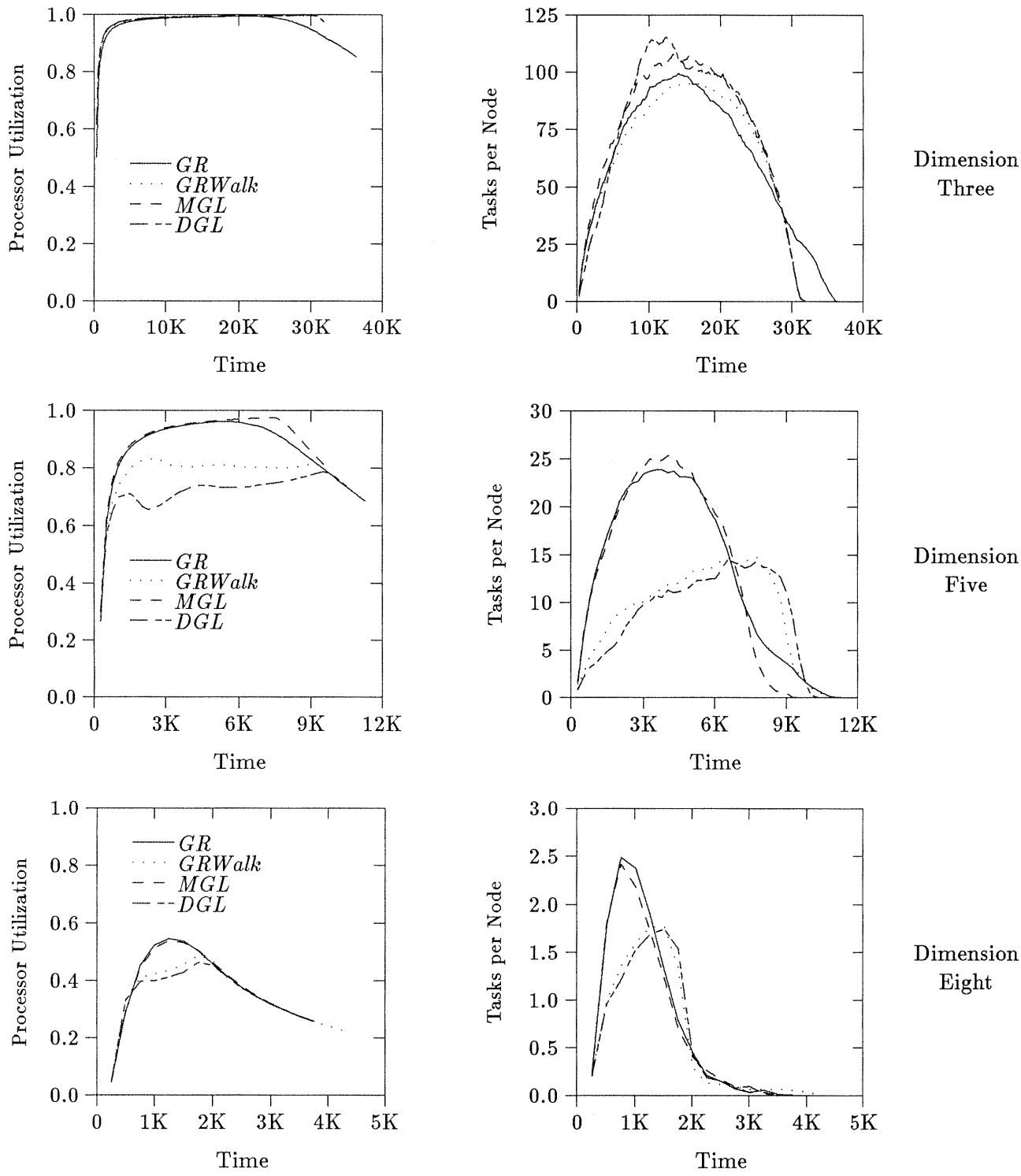


Figure 10: `Minmax` Workload: Time Spent Waiting for Messages

maximum possible speedup for `Minmax` (i.e., it is the minimum of the number of nodes and the maximum speedup shown in Table 4). As Figure 3 shows, the variance in speedup for a particular placement strategy can be substantial for hypercubes of reasonable dimension. Although additional simulation runs would reduce the width of the confidence interval, the high variance suggests that some strategies are particularly sensitive to the pattern of placement decisions.

Of the local strategies, *LR* is clearly inferior to *DLL* and *MLL*, especially for larger system sizes. Although speedups for the local strategies suggest that the **Random** policy is inferior, the speedups for global strategies imply otherwise; all global strategies, including *GR*, the global random strategy, have roughly equal performance.

The processor utilization, shown in Figure 4, confirms that those strategies with higher speedups also have higher mean processor utilizations. As with the speedup data, processor utilization is a coarse measure that does not reflect time varying resource demands, nor does it reveal the dynamics of placement policies — there are many possible ways to place tasks in both space and time that



(a) Processor Utilization

(b) Mean Tasks per Node

Figure 11: `Minmax` Workload: Processor Utilization and Task Distribution

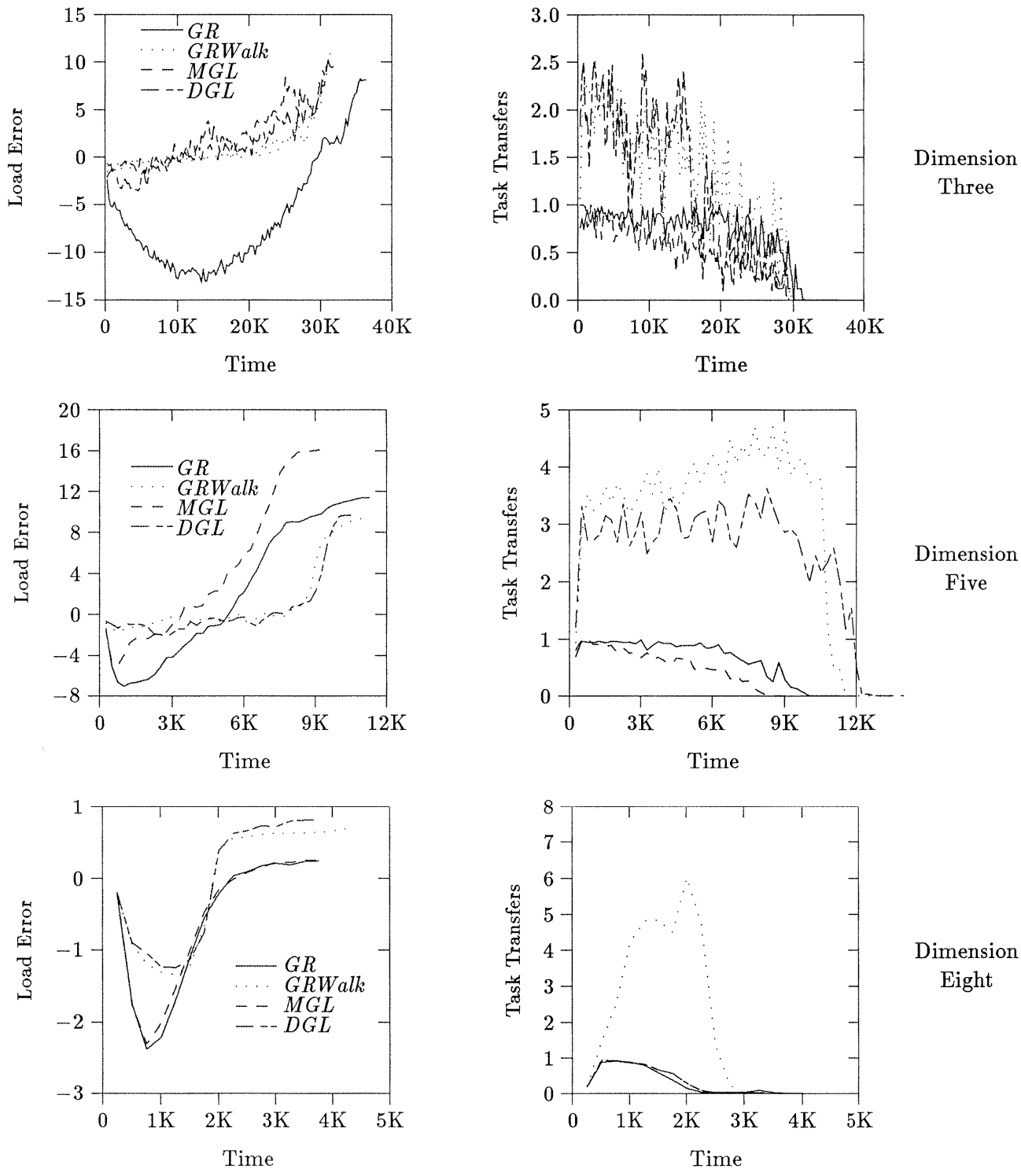


Figure 12: `Minmax` Workload: Load Error and Number of Placement Transfers

have similar speedups and processor utilizations. Figure 5, which shows the number of transfers needed to place a task, illustrates precisely this point.

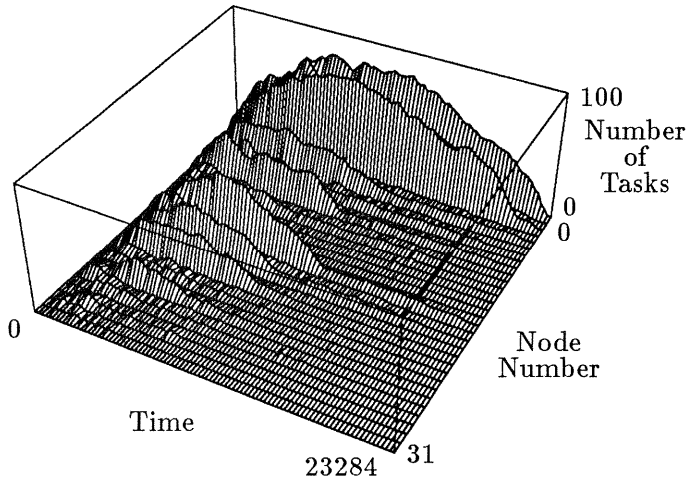
Both the *DLL* and *DGL* strategies repeatedly transfer tasks using load information, and in both cases, the maximum number of transfers is, by design, bounded by the hypercube dimension. Table 6 and Figure 3 show that *DLL* and *DGL* have qualitatively similar speedups, though the speedup of *DLL* is slightly higher. However, Figure 5 shows that *DLL* does not transfer tasks as frequently as *DGL* for larger system sizes. Recall that the *DLL* strategy maintains load information only about neighboring nodes; many local task transfers may be needed to reach a distant, least loaded node. Moreover, with only local information, we believe the *DLL* strategy is more likely than *DGL* to be trapped in local minima.

In contrast to *DLL*, the *DGL* strategy maintains load information on *all* nodes. However, Figure 12 indicates that this information is inaccurate. Following the initial burst of task creation in `Minmax`, most nodes underestimate the remote load — information on the change of state for distant nodes has not yet propagated across the network. Simultaneously, those nodes with tasks to distribute, observe, accurately, that they are overloaded. Based on a comparison of their local estimate of high load with the incorrect estimate of low remote load, these nodes select another, ill-informed node. This process repeats until sufficient task transfers accrue to reach the transfer limit. As the system size increases, remote load information is increasingly accurate, not because information dissemination improves, but because more nodes truly are lightly loaded. After the initial set of tasks has been placed, the aggregate performance is a *fait accompli*, determined by the initial actions.

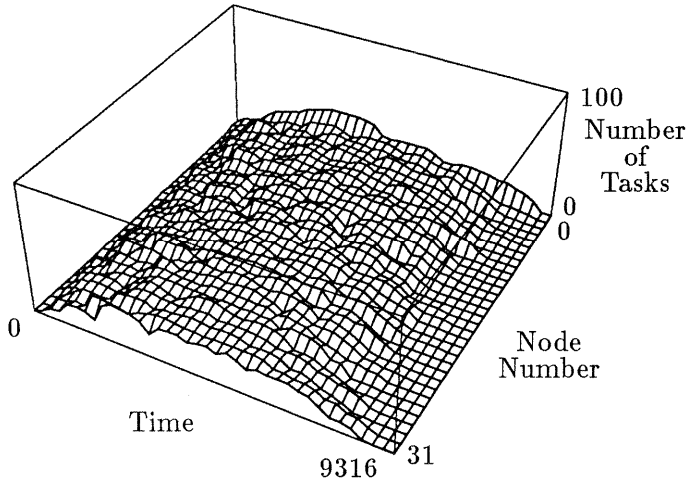
Although both *DGL* and *DLL* transfer tasks, and, on average, *DGL* forwards tasks much more than *DLL*, both have similar performance. Furthermore, their performance is similar to strategies such as *MGL*, *MLL* and *GR* that transfer tasks at most once. One might hope that the *DGL* and *DLL* strategies have some other, less obvious, redeeming benefit. For example, perhaps their strategies yield better communication locality, reducing the total communication time. However, Figures 7 and 8 show that *DLL* and *DGL* greatly increase the communication overhead.

Why, then, do these strategies perform well? Figure 9 shows the average number of tasks queued for the computation processor on each node, and Figure 6 shows the time needed to place a task. In small and moderate sized systems, a large number of tasks must wait for a computation processor. Although additional task transfers increase both the average task placement time and the expected message transmission time, shown in Figure 7, the total volume of computation so greatly exceeds available computation resources that the overhead for task transfers makes little difference. Simply put, excess tasks can either be queued for an available processor or shuttled through the network until the computation backlog is satisfied; neither option affects processor utilization. For larger systems, the computation processor queues are short, and the communication traffic is distributed across a large number of nodes. This decreases the message transmission time and ameliorates the effect of repeated task transfers; although tasks are transferred frequently, it costs little.

For the `Minmax` program, strategies that use load information, such as *DLL* and *DGL* have no special benefit; they simply avoid the load inequities of *LR*. Figure 11 illustrates this clearly for the global placement strategies. Recall that the *GRWalk* strategy is similar to *DGL*. A task is sent to a node; if that node believes others nodes are less busy, the task is transferred again. However, *GRWalk* transfers the task to a *random* node; no non-local load information is used. This process repeats until a node with sufficiently low load is found or until the number of transfers equals the dimension of the network. In Figure 11, the *GRWalk* and *DGL* strategies clearly form

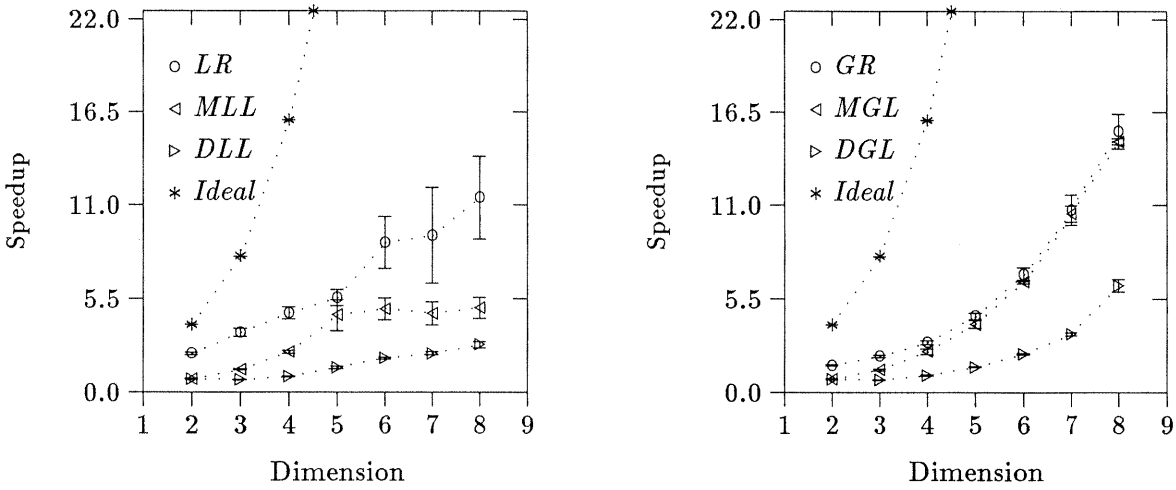


(a) Task Distribution for *LR* Strategy (Five-dimensional Hypercube)



(b) Task Distribution for *MLL* Strategy (Five-dimensional Hypercube)

Figure 13: `Minmax` Workload: Time Varying Number of Tasks per Node



(a) Local Strategies

(b) Global Strategies

Figure 14: Speedup Comparison For `Minmax` Workload: High-Latency Network

one equivalence class, with *GR* and *MGL* in another. The latter two strategies transfer tasks at most once. Obviously, the potential benefit of global load information is critically dependent on its accuracy. Figure 12 suggests that for large systems, global information is most often inaccurate — information driven strategies degenerate to random walks. In such cases, informationless strategies have comparable or superior performance with lower overhead.

The poor performance of the local random strategy *LR* demonstrates both the importance of widely dispersing tasks when the total number of tasks is high and the relative unimportance of communication locality in high-speed, circuit-switched networks. Global methods can place tasks anywhere in the network, whereas local methods require multiple transfers to achieve a similar effect. Only the *LR* strategy limits task placement to immediately adjacent nodes. The *task depth* of `Minmax`, shown in Table 4, is four, meaning the most distant relative of the original task is only four generations removed. Tasks of each successive generation can be placed one hop further from the initial parent. Because a binary hypercube with diameter n has $\binom{n}{k}$ nodes exactly k hops away, the *LR* strategy can place the tasks of `Minmax` on at most $\sum_{k=0}^{4}\binom{n}{k}$ nodes. The *MLL* strategy can cross twice the number of links, making effective use of more nodes. This is dramatically illustrated by recording the number of tasks assigned to each node as a function time; see Figure 13, which shows the time varying number of tasks on each node of a five-dimensional hypercube.

Based on our analysis of the `Minmax` program, we conclude that load information often is not particularly useful (i.e., its accuracy is too low), a large number of task transfers may not improve performance, and strictly local strategies such as *LR* are not effective. One may well ask how closely these conclusions are tied to the network architecture. To investigate the interaction of network performance, workload, and placement strategy, we increased the message transmission latency from ten to one hundred microseconds, while holding the bandwidth constant at thirty-two megabytes per second. Comparing Figures 3 and 14 shows that, not surprisingly, speedup declines for all strategies. However, the performance of the local strategies is comparatively poorer than before.

Based on our earlier analysis, the reasons for performance shifts should be clear. Higher communication latency penalizes those strategies that must transfer tasks many times (i.e., *DLL* and *DGL*), regardless of the distance of each transfer. Thus, one would expect single hop strategies (i.e., *LR* and *GR*) to fare well. *GR* remains the strategy of choice; it uses only a single transfer and, in contrast to *LR* can place tasks on any node. For the local strategies, *LR* has a high probability of placing tasks on the current node, reducing the number of messages that must be sent to other nodes. Although the *MLL* strategy can offload tasks to more nodes than *LR*, it always sends tasks to the least loaded node, causing most tasks to be distributed, although only to nearby nodes. This is exacerbated by the shallow task tree of the `Minmax` program. Simply put, *LR* has poorer load balance, but this is an advantage at high latency — minimizing communication, not maximizing load balance, is most important when the costs of load balancing are high relative to task lifetimes.

For either global or local methods, a threshold mechanism favoring the current node becomes increasingly critical as network latency increases. Moreover, the global methods perform relatively better than the local methods because circuit-switched and wormhole networks diminish the importance of spatial locality.

6.4 Case Study: `MultiGrid`

As we have seen, the `Minmax` program is a highly parallel application with many short lived tasks that are created near the beginning of program execution. In this section, we examine a `MultiGrid` program with strikingly different behavior. Like `Minmax`, most of the tasks are created near the beginning of execution. However, the pattern of task creation is different — the tasks of `Minmax` are recursively created, forming a broad, shallow, balanced tree of task ancestry. In contrast, two `MultiGrid` tasks are the parents of over half of all the other tasks. Also, each `MultiGrid` task represents considerable computation with many large messages sent to and received from other tasks. Table 4 shows that the potential concurrency is relatively lower than that for `Minmax`, but it is highly variable. Tasks are activated cyclically, as the multigrid algorithm refines the mesh, causing tasks to exchange messages.

The characteristics of the `MultiGrid` program present a significant challenge to load distribution strategies. Figure 15 shows that the speedups for the *LR* and *MLL* strategies are relatively poor. As with the `Minmax` program, the *LR* strategy is limited by the task depth of the `MultiGrid` program. When a single task creates many children, the *LR* strategy can place these tasks only on nodes adjacent to the node of the parent.

The *MLL* strategy also suffers from the task depth; tasks can be placed up to two hops away, effectively using only a fraction of the available nodes. However, the `MultiGrid` tasks, unlike those of `Minmax` are long lived, making *MLL*'s limited task dispersion debilitating.

Both *MLL* and *MGL* select nodes using an estimated load value supplied by the destination node. When a task is enqueued for transfer to another node, the estimated load for the remote node is locally incremented to reflect the new task. Initially, this is a good approximation; however, when a node actually begins the task transfer, that node's estimate of the remote node's load is replaced with the latest information from the remote node. In the `MultiGrid` program, most tasks are created early in the computation. Because of this, the load estimates obtained from remote nodes are too low because they do not yet account for tasks enroute. Thus, certain nodes are erroneously, and repeatedly selected. Figure 24 shows that task placement occurs in three distinctive stages. Estimates of remote load are not updated sufficiently rapidly to supply useful

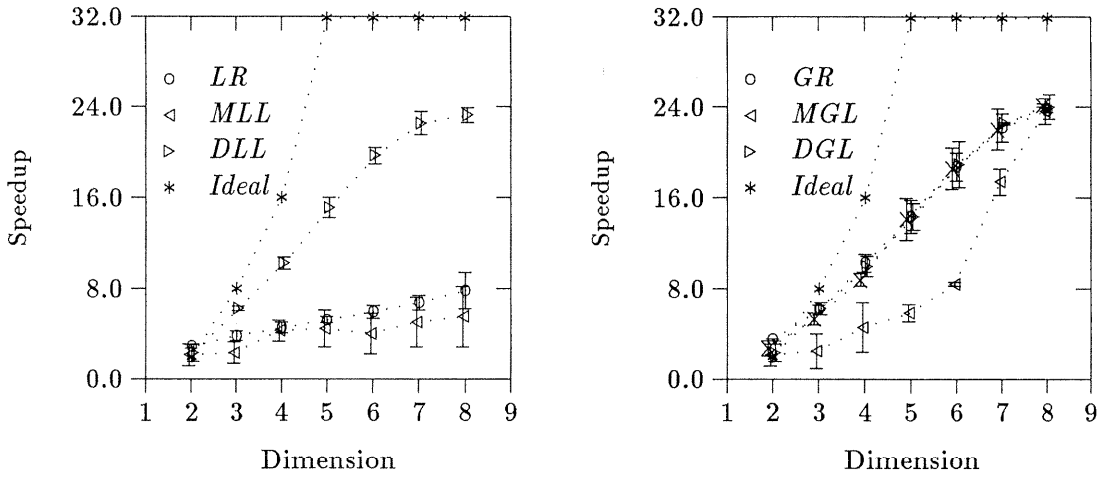


Figure 15: `MultiGrid` Workload: Speedup

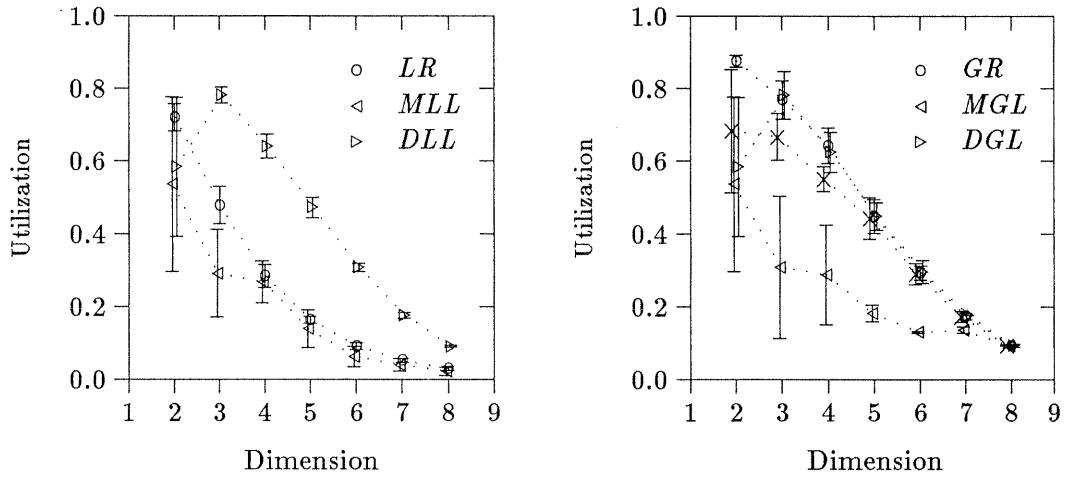


Figure 16: `MultiGrid` Workload: Processor Utilization

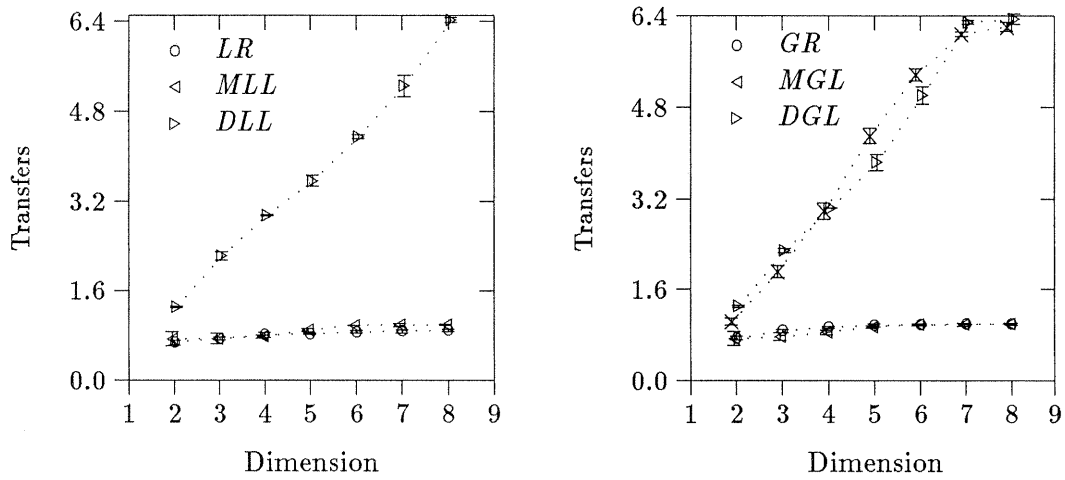


Figure 17: `MultiGrid` Workload: Number of Transfers Per Placement

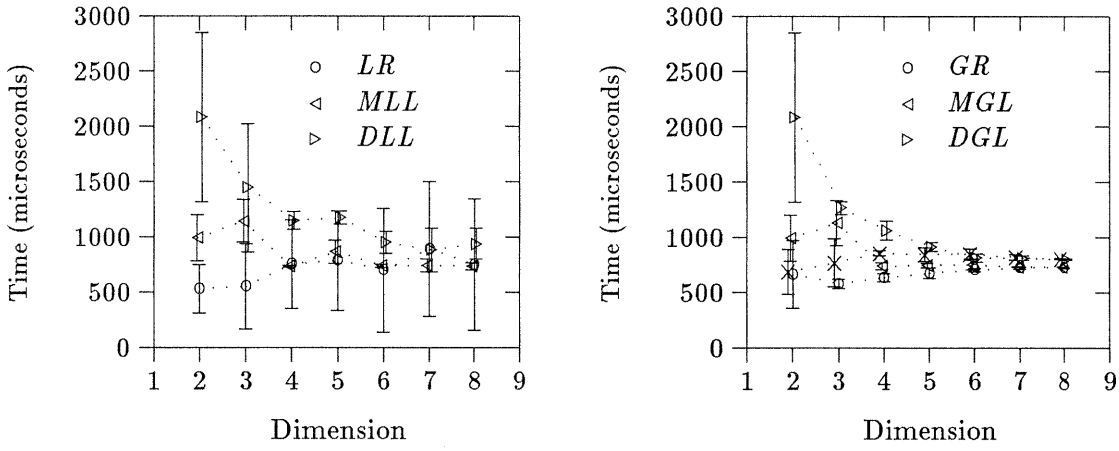


Figure 18: `MultiGrid` Workload: Time Spent Finding a Destination Node

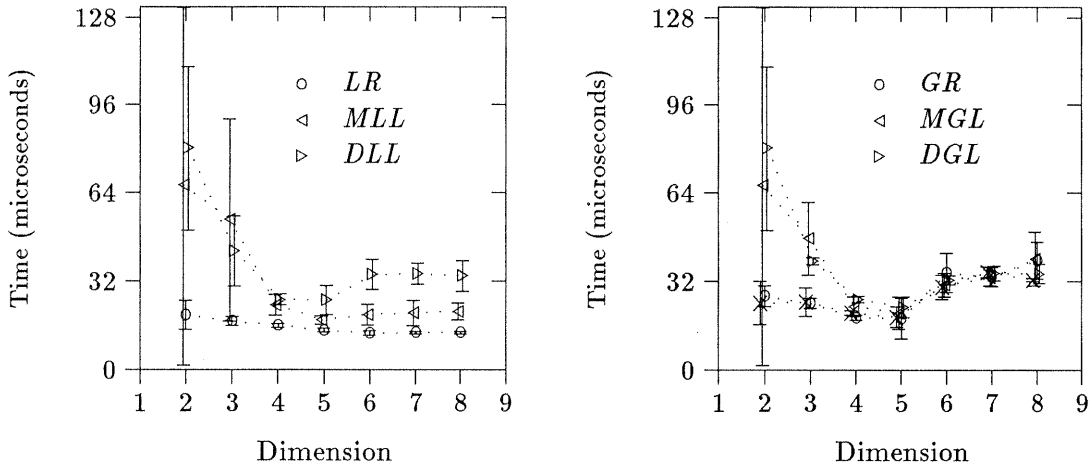


Figure 19: `MultiGrid` Workload: Average Message Transmission Time

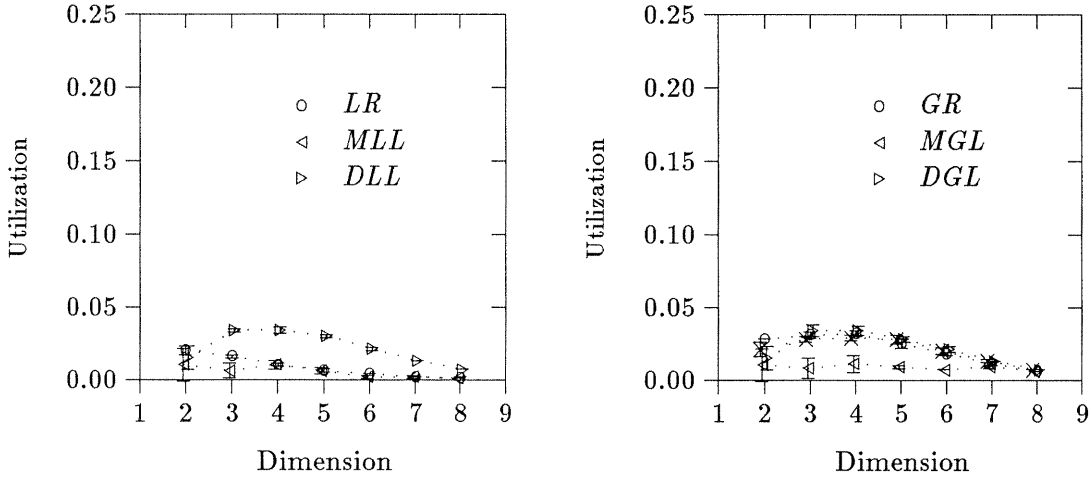


Figure 20: `MultiGrid` Workload: I/O Processor Utilization

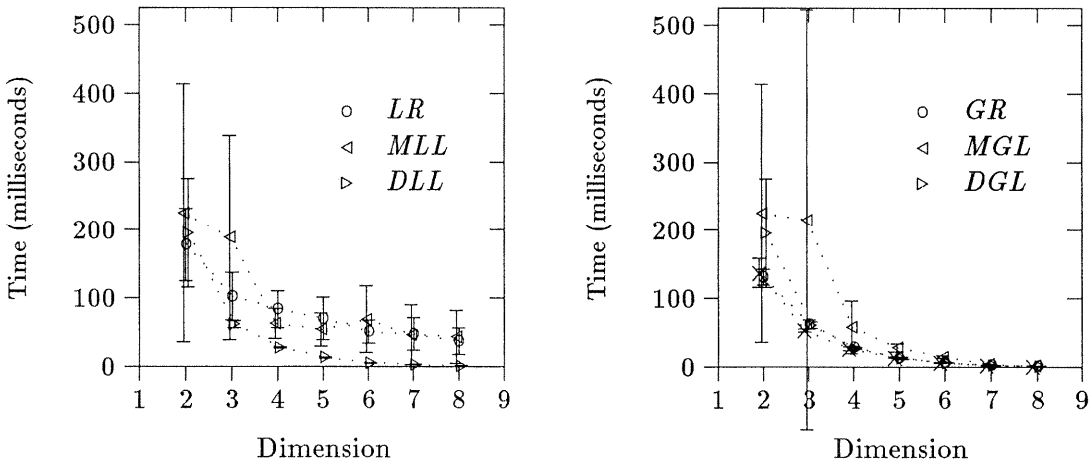


Figure 21: `MultiGrid` Workload: Time Spent Waiting for Processor

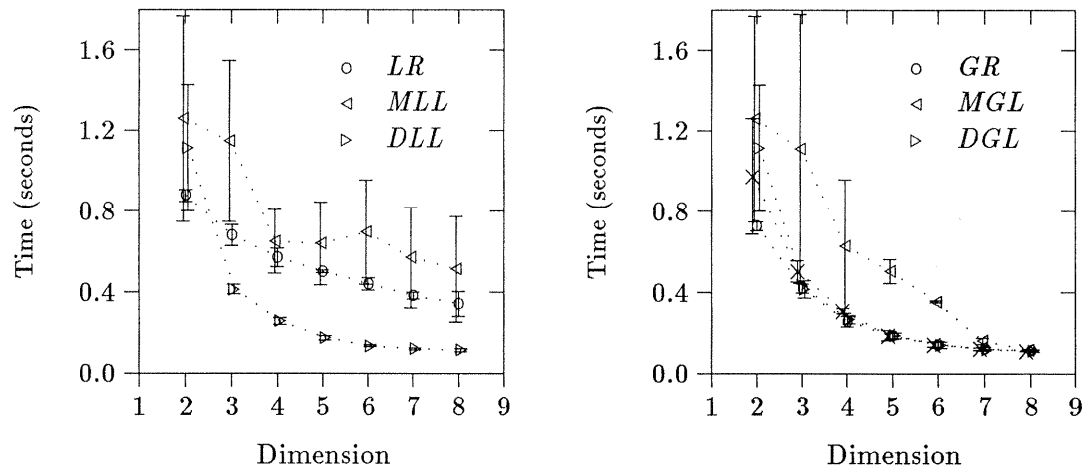
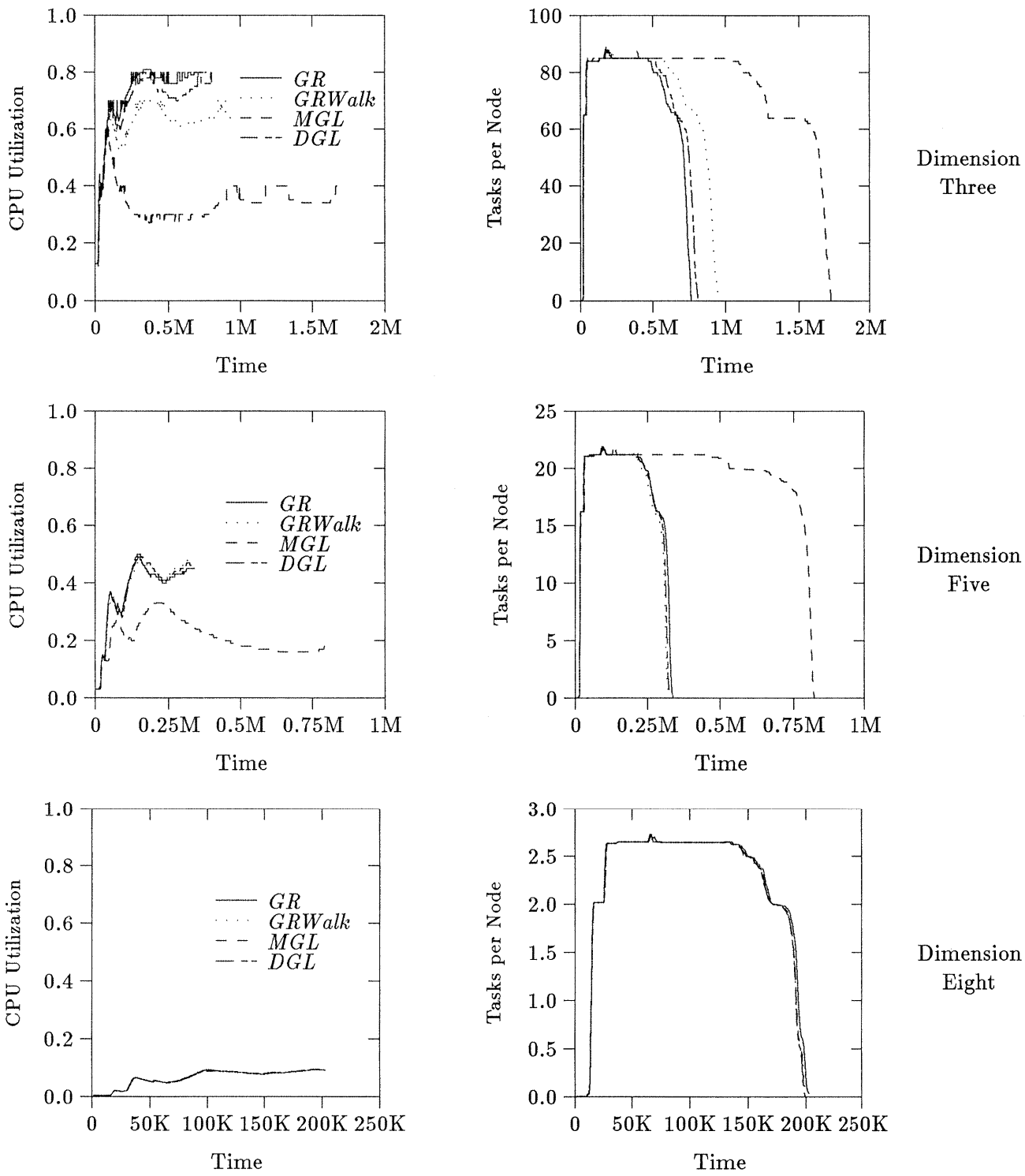


Figure 22: `MultiGrid` Workload: Time Spent Waiting for Messages



(a) Processor Utilization

(b) Mean Tasks per Node

Figure 23: `MultiGrid` Workload: Processor Utilization and Task Distribution

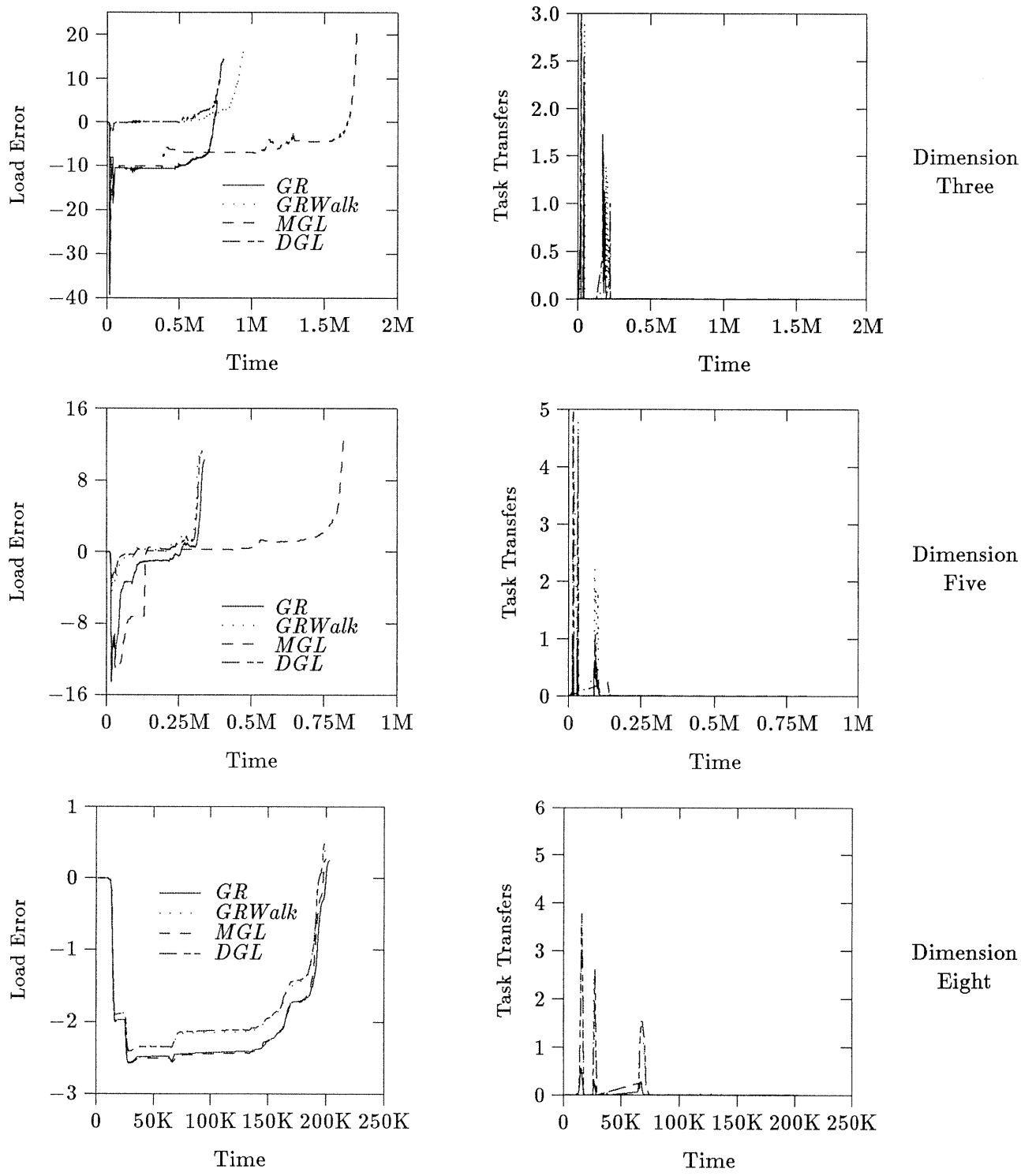


Figure 24: `MultiGrid` Workload: Load Error and Placement Transfers

information. Individual, simulated instances of the *MGL* strategy show that as few as two nodes receive the majority of the tasks.

The problem of inaccurate load information dissemination is exacerbated by the network architecture. Transiting messages impart no information to intermediate nodes; only the source and destination of each message acquire the load information stored in the message. One of us is examining the benefit of augmenting the the network architecture to impart this information to intermediate nodes [?]. Although it is unlikely that nodes could accurately record the status of all nodes for large systems, they may accurately track the status of topologically adjacent nodes, making strategies such as *GRD.T* viable.

Figure 17, which shows the number of task transfers, is similar to Figure 5 for `Minmax`. However, the *time* spent placing a task, Figures 6 and 18, differs dramatically. For `MultiGrid`, all strategies, including those that transfer tasks only once, encounter considerable delay when placing tasks. As we have seen, most tasks are the children of two parents. This serialized task creation means that a few tasks quickly create a large number of child tasks that must be enqueued for distribution by the communication processors of the nodes executing the parent tasks. Because the task placement delay includes the queueing delay for distribution, the mean delay in Figure 18 is high for all placement methods.

We noted earlier that `MultiGrid` tasks interact many times during their execution, not just with their parent at task termination. Moreover, the average `MultiGrid` message is a thousand times the size of a `Minmax` message. The latter explains the higher message transmission time shown in Figure 19. However, the large message waiting time, shown in Figure 22, is *not* caused by high communication processor utilization, as Figure 20 shows. Instead, it is a function of both task interactions and task placement. A task's delay while awaiting a message may be caused by the sending task's contention for a computation processor on a remote node. Only after the sending task acquires its local processor and executes sufficient code to reach the message transmission event can the message be enqueued for transmission by the communication processor (i.e., processor waiting can induce message waiting). The dependency chain of message waiting delays may include multiple tasks. The larger computation granules of the `MultiGrid` tasks, reflected in the larger processor waiting times of Figure 21 is a contributing factor to larger message delays. Thus, load balance is insufficient to minimize message waiting delays.

Analysis of the `MultiGrid` program reinforces the lessons gleaned from the `Minmax` program, namely that informationless strategies such as *GR* are robust when placing widely varying workloads and that the load information used to place tasks must be accurate; ideally it should be augmented with information concerning tasks *enroute* to other nodes. Finally, no placement method can quickly disperse tasks if the task creation tree is degenerate. This degeneracy can be corrected in several ways. The simplest requires the programmer or language system to impose a divide-and-conquer task creation mechanism, if permitted by the task semantics. However, the crux of the problem is limited message transmission capability, even though our simulated architecture includes aggressive assumptions about message latency and bandwidth.

A better solution is *load redistribution* via *task migration.* Typically, task migration is used to balance processor load. However, consider the benefits of balancing *communication* performance as well. If the `MultiGrid` tasks that are the progenitors of most other tasks were migrated to other nodes after creating a number of tasks, they could continue to execute, creating additional tasks there. Such a migration would effectively multiply the task placement rate because additional nodes could now participate.

7 Conclusions

It is foolhardy to draw strong conclusions from a small sample size; however, we purposefully varied the parameters of our sample workloads and simulated architecture to cover a broad range of behavior and possible designs. From our studies, we believe effective workload distribution strategies should place new tasks globally, rather than locally, to spread tasks rapidly. This is particularly important with the high latency networks in common use, where repeatedly transferring tasks consumes considerable resources. We also believe informationless strategies are competitive with those that rely on distributed load metrics; this is particularly true if the time constant for information propagation is greater than the expected time between gross changes in the system state.

As part of our ongoing study, we are examining task *migration* strategies that respond to a variety of resource constraints, including processor utilization and message transmission delay. We hope such strategies can refine a coarse task distribution, produced by an initial task placement strategy (e.g., one of the strategies described in this paper), using information about task dynamics to move tasks after they have begun execution. This is particularly important in the absence of *a priori* information on task lifetimes and interactions; in these cases, few placement methods will provide satisfactory performance. In addition, we are exploring new network topologies, including two-dimensional meshes, and we are examining task placement algorithms for heterogeneous systems.

Finally, we believe a detailed examination of *system dynamics* is crucial to understanding the performance of task distribution methods. We have found it exceedingly difficult to infer the reasons for poor or highly variable performance without this information. By their nature, these transient characteristics are difficult to model and can dramatically affect performance. Only by studying actual programs will we be able to design and refine effective load distribution methods.

References

[1] AMETEK. *The Ametek System 14 Users Guide.* Ametek Computer Research Division, May 1986. Document No. V12970.

[2] ARLAUSKAS, R. iPSC/2 System: A Second Generation Hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I* (Pasadena, CA, January 1988), ACM, pp. 38–42.

[3] ATHAS, W., AND SEITZ, C. Cantor User Report, Version 2.0. Tech. Rep. 5232:TR:86, California Institute of Technology, Department of Computer Science, January 1987.

[4] ATHAS, W. C., AND SEITZ, C. L. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer* (August 1988), 9–24.

[5] CARRIERO, N., AND GELERNTER, D. Linda in Context. *Communications of the ACM 32*, 4 (April 1989), 444–458.

[6] CARROLL, M. A Comparison of Two Dynamic Load Balancing Schemes for Multiprocessor Systems. Master's thesis, University of Illinois at Urbana-Champaign, 1987.

[7] CHIEN, A. A., AND DALLY, W. J. Experience with Concurrent Aggregate (CA): Implementation and Programming. In *Proceedings of the* 5^{th} *Distributed Memory Computing Conference* (April 1990), Association For Computing Machinery, pp. 1040–1049.

[8] CHOW, E., MADEN, H., PETERSON, J., GRUNWALD, D., AND REED, D. Hyperswitch Network for the Hypercube Computer. In *Proceedings of the* 15^{th} *International Symposium on Computer Architecture* (June 1988), pp. 90–99. to appear in *IEEE Transactions on Computers.*

[9] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation 6*, 1 (March 1986), 53–68.

[10] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. Adaptive Load Sharing in Homogenous Distributed Systems. *IEEE Tranactions on Software Enginerring SE-12*, 5 (May 1986), 662–675.

[11] GRABAS, D. Design and Evaluation of a Performance Data Capture Program. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1989.

[12] GRIMSHAW, A. The Mentat Run-Time System: Support for Medium Grain Parallel Computation. In *Proceedings of the* 5^{th} *Distributed Memory Computing Conference* (April 1990), Association For Computing Machinery, pp. 1064–1073.

[13] GRUNWALD, D. C. *Heuristic Load Distribution in Circuit Switched Multicomputer Systems.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1989.

[14] GRUNWALD, D. C., AND REED, D. A. Analysis of Backtracking Routing in Binary Hypercube Computers. Tech. Rep. UIUCDCS-R-89-1486, University of Illinois at Urbana-Champaign, November 1987.

[15] GRUNWALD, D. C., AND REED, D. A. Benchmarking Hypercube Hardware and Software. In *Hypercube Multiprocessors* (1987), M. Heath, Ed., Society for Industrial and Applied Mathematics, pp. 169–177.

[16] GRUNWALD, D. C., AND REED, D. A. Networks for Parallel Processors: Measurements and Prognostications. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I* (Pasadena, CA, January 1988), ACM, pp. 610–619.

[17] HAYES, J. P., MUDGE, T., STOUT, Q. F., COLLEY, S., AND PALMER, J. A Microprocessor-based Hypercube Supercomputer. *IEEE Micro 6*, 5 (October 1986), 6–17.

[18] KALÉ, L. V. *Parallel Architectures for Problem Solving.* PhD thesis, State University of New York at Stony Brook, December 1985.

[19] KALÉ, L. V. Comparing the Performance of Two Dynamic Load Distribution Methods. Tech. Rep. UIUCDCS-R-87-1776, University of Illinois at Urbana-Champaign, Department of Computer Science, November 1987.

[20] KALÉ, L. V., AND SHU, W. The Chare-Kernel Language for Parallel Programming: A Perspective. Tech. Rep. UIUCDCS-R-89-1451, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1989.

[21] KELLER, R., LIN, F., AND TANAKA, J. Rediflow Multiprocessing. In *CompCon '84* (1984), pp. 410–417.

[22] KRATZER, A., AND HAMMERSTROM, D. A Study of Load Leveling. In *Compcon '79* (Fall 1980), IEEE, pp. 647–654.

[23] LILLEVIK, S. L. Touchstone Program Overview. In *Proceedings of the* 5^{th} *Distributed Memory Computing Conference* (April 1990), Association For Computing Machinery, pp. 647–657.

[24] LIN, F. C. H. *Load Balancing and Fault Tolerance in Applicative Systems.* PhD thesis, University of Utah, June 1985.

[25] LIN, F. C. H., AND KELLER, R. M. The Gradient Model Load Balancing Method. *IEEE Software SE-13*, 1 (January 1987), 32–38.

[26] NAZIEF, B. A. A. *Load Distribution in Multicomputer Systems.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, Sept. 1991.

[27] RAMKUMAR, B., AND KALÉ, L. V. Compiled Execution of the Reduce-Or Process Model on Multiprocessors. Tech. Rep. UIUCDCS-R-89-1513, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1989.

[28] RATTNER, J. Concurrent Processing: A New Direction in Scientific Computing. In *Proceedings of the 1985 National Computer Conference* (1985), AFIPS Press, pp. 157–166.

[29] REED, D. A., AND FUJIMOTO, R. M. *Multicomputer Networks: Message-Based Parallel Processing.* The MIT Press, Cambridge, Mass., 1987.

[30] SEITZ, C. L., ATHAS, W. C., FLAIG, C. M., MARTIN, A. J., SEIZOVIC, J., STEELE, C. S., AND SU, W.-K. The Architecture and Programming of the Ametek Series 2010 Multicomputers. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I* (May 1988), ACM, pp. 33–36.

[31] SHU, W., AND KALÉ, L. V. Dynamic Scheduling of Medium Grained Processes on Multiprocessors. Tech. Rep. UIUCDCS-R-89-1528, University of Illinois at Urbana-Champaign, Department of Computer Science, November 1989.

[32] SULLIVAN, H., AND BRASHKOW, T. A Large Scale Homogeneous Machine I & II. In *Proc. 4th Annual Symposium on Computer Architecture* (1977), pp. 105–124.

[33] TILBORG, A. V., AND WITTIE, L. D. Wave Scheduling – Decentralized Scheduling of Task Forces in Multicomputers. *IEEE Transactions on Computers C-33*, 9 (September 1984), 835–864.

[34] VAN TILBORG, A., AND WITTIE, L. D. Distributed Task Force Scheduling in Multi-Microcomputer Networks. In *Proceedings of the 1981 National Computer Conference* (1981), AFIPS, pp. 283–289.