AD-A278 789

RL-TR-94-7
Final Technical Report
March 1994

TASK ALLOCATION FOR PARALLEL REAL-TIME EXECUTION

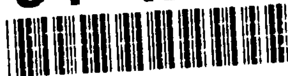
Calspan - UB Research Center

John K. Antonio and Richard C. Metzger

DTIC
ELECTE
MAY 02 1994
S G D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

94-12982



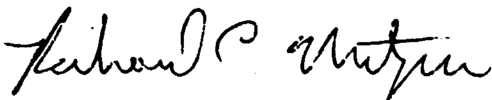
Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

94 4 28 058

Best
Available
Copy

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-7 has been reviewed and is approved for publication.

APPROVED: 

RICHARD C. METZGER
Project Engineer

FOR THE COMMANDER:

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1994	Final Jun 93 - Aug 93

4. TITLE AND SUBTITLE
TASK ALLOCATION FOR PARALLEL REAL-TIME EXECUTION

5. FUNDING NUMBERS
C - F30602-93-D-0075, Task 0006
PE - 61102F
PR - 2304
TA - F2
WU - 01

6. AUTHOR(S)
John K. Antonio and Richard C. Metzger

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Calspan-UB Research Center
P O Box 400
Buffalo NY 14225

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CB)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-94-7

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Richard C. Metzger/C3CB/(315) 330-7650
John K. Antonio-Purdue University Richard C. Metzger-Rome Laboratory

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
Current trends indicate that future C3 (command, control, and communications) systems will likely contain suites of heterogeneous computing facilities composed of both parallel and sequential computing components. Also, "commercial-off-the-shelf" components are being heavily considered for use in these future systems. In the context of these trends, brief overviews of related work within the areas of parallel processing and real-time computing are given and areas of future research are outlined. A central theme throughout the report is that in order to make effective use of future C3 platforms, a significant amount of "cross fertilization" between researchers in the parallel processing and real-time computing communities will be required. As an example of the need for combined expertise in both areas, the problem of how to effectively allocate periodic real-time tasks onto the processing elements of a hypercube architecture is illustrated through an example. Other research issues, including task partitioning, operating systems, I/O, and the software development process are also discussed.

DTIC QUALITY INSPECTED 3

14. SUBJECT TERMS
Static Task Allocation, Hypercube, Periodic Tasks

15. NUMBER OF PAGES
40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Executive Summary

One of the fundamental problems with the integration of parallel computing platforms into real-time embedded systems, is the scheduling of the software components to guarantee the absence of deadline violations. Given parameters such as the periodicity of the tasks, interconnection architecture, and communication bandwidth can play a vital role in how the tasks are mapped onto processors. The focus of this effort was to review current static task allocation techniques and determine their applicability for allocating periodic tasks in a hypercube system. During the course of the effort, the mapping techniques were simulated against an air defense model adapted from the literature. The simulation was carried out at Rome Laboratory using the Optimal Mapping Alternate Routing System (OMARS) which was developed via a joint in-house Rome Laboratory/Purdue University effort.

Accesion For
NTIS CRA&I X
DTIC TAB ☐
Unannounced ☐
Justification

By
Dist ibution /

Availability Codes

Dist | Avail and / or Special
A-1

I. INTRODUCTION

With rapid advances being made in technology for information acquisition for C3 systems, the Department of Defense has recognized the need for extensive research in high-performance real-time information processing. The Global Surveillance and Communications thrust is an example of an area that requires technology for acquiring information from remote sensors, transmitting information from remote senors to a command and control center, and processing information so that orders can be issued to investigate and/or counter threats. These command and control centers for Global Surveillance and Communications Systems may be ground based or airborne.

In general, command and planning centers must have the computing power to make sense of incoming information, respond to new threats, and plan missions to counter threats. Recently, much of the required processing power has come in the form of distributed networks of workstations. Advantages of using networks of workstations include the relative ease of programming, well supported operating systems, and a large user base in the industrial sector. Because of these attributes, a substantial amount of high-quality and portable software has been developed for workstation environments.

As high-performance parallel processing systems become more suited physically (in terms of size and ruggedness), they too will find their way into operational use in command and control centers. Some airborne systems are already being developed such as the nCUBE/Microlithics hypercube [9]; also, the Advanced Research Projects Agency (ARPA) has recently sponsored an effort with Intel and Honeywell to produce an militarized version of the Touchstone machine. Unfortunately, a systematic software development process for such machines, especially real-time information processing systems, is still largely undefined. One difficulty associated with developing real-time software for parallel systems is that the degree of nondeterminism in (some) parallel systems makes it difficult to determine schedules for resources such as the processors and memory modules.

This nondeterministic timing problem is in direct conflict with the inherently time-critical nature of real-time C3 applications.

The types of real-time computations required by C3 systems cover a wide spectrum: the deadlines for completing a computational task may be soft or hard and the repetition of the computational tasks may be periodic or asynchronous. A deadline for a real-time task is said be a hard deadline if completing the computation after the deadline is of no value and/or results in catastrophic consequences. For instance, from the time an incoming enemy missile is first detected, the computation required to effectively counter the attack must be delivered before a critical time deadline, i.e., before the missile enters a critical zone. A task with a soft deadline implies that while completing the computation before the deadline is desirable, occasionally missing the deadline can be tolerated [20]. Examples of computational tasks that may have soft deadlines include certain image/speech processing tasks, statistical processing of archived data, and data-base queries.

A periodic task is a task for which the repetition of its execution is known and fixed. Aircraft flight control systems and flight simulators require the computation of periodic real-time tasks, e.g., in a flight control system, the task of computing the control signal for driving an actuator of a control surface is a periodic task. It has been observed that a majority of the computational tasks in today's weapons systems are of the periodic type [20]. An asynchronous task is one in which the demands for its execution are either unknown or unpredictable. Certain tasks within C3 that are activated by sensory information, e.g., the "determination of 'friend or foe' of an approaching aircraft," could be classified as asynchronous real-time tasks.

The remainder of the report is organized in the following manner. In Section II, brief overviews of related research in the areas of parallel processing and real-time computing are given. In Section III, the problem of mapping tasks onto the processors of a hypercube

architecture (as generally defined by the parallel processing community) is reviewed. Section IV introduces a new research problem—how to effectively map periodic real-time tasks onto the hypercube architecture—as a means of demonstrating the importance of having expertise from both the parallel processing and real-time computing areas when using parallel systems for real-time applications. In Section V, some general research issues are briefly outlined in the context of how to effectively use parallel processing systems for real-time applications. A summary of the report and some concluding remarks are included in the last section.

II. Related Research

Brief overviews of related work in the areas of both parallel processing and real-time computing are included in this section. These overviews include only a representative subset of important topics from within each area and should not be viewed as complete tutorials. The overviews are presented to demonstrate the range of issues that must be considered in order to make effective use of parallel processing platforms for use in real-time applications. Just as the discussions within the overviews contain only a representative subset of important topics from within each area, the cited references for the various topics are also only a representative few.

A. Parallel Processing

The basic concept of parallel processing is to perform computation by first decomposing a given task into subtasks and then executing these subtasks on independent processors. The goal is to decrease the total time required to complete the overall task by simultaneous execution of the subtasks. While the philosophy of parallel processing is clear, and while there are both practical and theoretical success stories, there are still many open questions. To illustrate the spectrum of unresolved issues that exist in parallel process-

ing, consider the following representative list of questions. To a certain degree, each of these questions is associated with an active area of research.

- How much parallelism is present within any given problem domain?
- Should parallelism be exploited early on during the initial design of the software (i.e., before any code is written)?
- Can general sequential programs be effectively parallelized using automated compiler techniques?
- How much parallelism should the user be expected to detect and exploit?
- How much parallelism should the compiler be expected to detect and exploit?
- What attributes should the language for parallel processing systems have?
- What system model should the programmer use in order to develop parallel software?
- What types of meaningful system models can be realistically implemented or approximated?
- What is the "best" general purpose parallel processing architecture?

With so many unresolved (and perhaps unresolvable) questions, one may ask "Why consider parallel processing?"and/or "Why not wait one year for the speed of sequential machines to double?"

The fact that the speed of light is a constant, and thus there is a definite upper bound on the attainable speed of sequential machines, provides some motivation for investigating the potential of parallel processing. Another more practical motivator involves economics. For certain application domains, some commercially available massively parallel processing systems have demonstrated cost/performance ratios superior to those of the more

traditional state-of-the-art vector supercomputers. Cray Computers Inc., perhaps the most successful manufacturer of vector machines, has recognized that massively parallel processors will eventually be the only way to continue to increase performance [14].

Perhaps the most popular taxonomy for parallel processors is the concept of the multiplicity of instruction stream and data stream, introduced by Flynn [8]. A single instruction stream - multiple data stream (SIMD) system executes the same thread of control (i.e., instruction stream) on all processors and each processor executes the instructions on distinct (locally stored) data. The term "SIMD" is often used synonymously with the term "data parallel." A multiple instruction stream - multiple data stream (MIMD) system is able to execute multiple independent threads of control, one or more on each processor. The term "MIMD" is often used synonymously with the terms "control parallel" and "functional parallel."

Machines that support the SIMD mode of parallelism typically have a special processor called the control unit that executes the single thread of control and broadcasts instructions to an array of processors that operate on there own data memories. Thus, the processors of an SIMD machine execute the same instruction at the same time (in lockstep synchronization) on distinct data. An example of a commercially available SIMD computer capable of supporting up to 16,384 processors is the MasPar MP-1 system, see [4] for a detailed description of its architecture.

Machines that support the MIMD mode of parallelism consist of an interconnected collection of independent processors, each capable of executing its own independent thread of control. An example of an MIMD machine capable of supporting up to 8,192 processors is the nCUBE 2 system [13].

It is generally acknowledged that the SIMD and MIMD modes of parallelism each have their own sets of advantages and disadvantages. For instance, it is usually agreed that SIMD machines are easier to program than MIMD machines; however, MIMD machines

allow more flexibility for expressing parallelism and thus offer the (potential) advantage of being suitable for a wider range of application domains.

Some experimental machines have been proposed and/or built that are capable of supporting both modes of parallelism. An example of such a "mixed-mode" machine is the PASM (*pa*rtionable *S*IMD/*M*IMD) system, which offers the advantage of being able to switch between modes of parallelism at instruction level granularity with negligible overhead [17]. Being able to change modes at a fine level of granularity is important in applications where several parts of the code are much better suited for one mode of computation than the other (i.e., SIMD versus MIMD).

A fundamental issue in the parallel processing area is the question of how to effectively map parallel algorithms onto parallel architectures. Because the interconnection networks employed by different machines have a wide range of characteristics and properties, the best mapping strategy for one architecture is generally not the best strategy for another. For more information on the various types of interconnection networks proposed and/or used in parallel processing systems, refer to [15]. To appreciate the range of strategies and techniques developed for mapping tasks onto parallel systems, compare the technique described in [16] with that described in [1].

B. Real-Time Computing

The basic concept of real-time computing is the scheduling of functionally distinct tasks to insure that deadlines are met. Failure to adhere to deadlines can lead to degraded system performance and/or loss of life or property.

There are two major approaches to solving the scheduling problem for real-time applications. The first is dynamic scheduling, where the scheduling of tasks is determined on-line as tasks arrive for execution [20]. The appeal of dynamic scheduling is that tasks can be scheduled for execution as a need arises and/or as resources become available in

the system. This may be especially useful in environments where there are many asynchronous real-time tasks to be executed. A critical issue in dynamic scheduling research is to devise fast scheduling algorithms so that the amount of time spent executing the scheduling algorithm does not infringe upon the performance of the system. Refer to [20, 21] for more detailed discussions of dynamic scheduling techniques.

A second scheduling approach is static scheduling, where the scheduling of tasks is determined off-line, before execution begins. This type of scheduling technique requires that certain attributes of the tasks be known in advance in order to compute a feasible schedule (if one exists). Static scheduling techniques are especially well-suited for environments where the majority of real-time tasks are periodic. Refer to [20] for more a detailed discussion of static scheduling techniques.

A well-known result for the scheduling of periodic tasks on a single processor system is the rate monotone algorithm [11]. The rate monotone algorithm is a preemptive, priority-driven algorithm, where priorities are assigned to tasks proportional to their rates of repetition. Given a collection of K periodic tasks with execution times and repetition periods denoted by $\underline{\tau_i}$ and $\underline{p_i}$, respectively, the rate monotone algorithm guarantees to provide a feasible schedule (i.e., no missed deadlines) provided that the following inequality is satisfied:

$$\sum_{i=1}^{K} \frac{\tau_i}{p_i} \leq K(2^{1/K} - 1). \tag{1}$$

The sum on the left-hand side of the equation represents the processor utilization. Each term in the sum is called the utilization factor, and is denoted by $\underline{\mu_i = \frac{\tau_i}{p_i}}$. It is easy to show that the right-hand side of the equation is bounded above by unity for all values of $K \geq 1$, and it approaches $\ln 2$ as $K \to \infty$.

To illustrate the rate monotone algorithm and its relationship to the above inequality, consider the examples shown in Figs. 1–3. In all three figures, the rate monotone algorithm is applied to a problem of scheduling three periodic tasks. The execution times and

$\tau_1 = 1 \quad p_1 = 3$
$\tau_2 = 2 \quad p_2 = 10$
$\tau_3 = 3 \quad p_3 = 15$

$$\sum_{i=1}^{3} \frac{\tau_i}{p_i} \le 3\left(2^{1/3} - 1\right)$$

$(0.7333 \le 0.7797)$

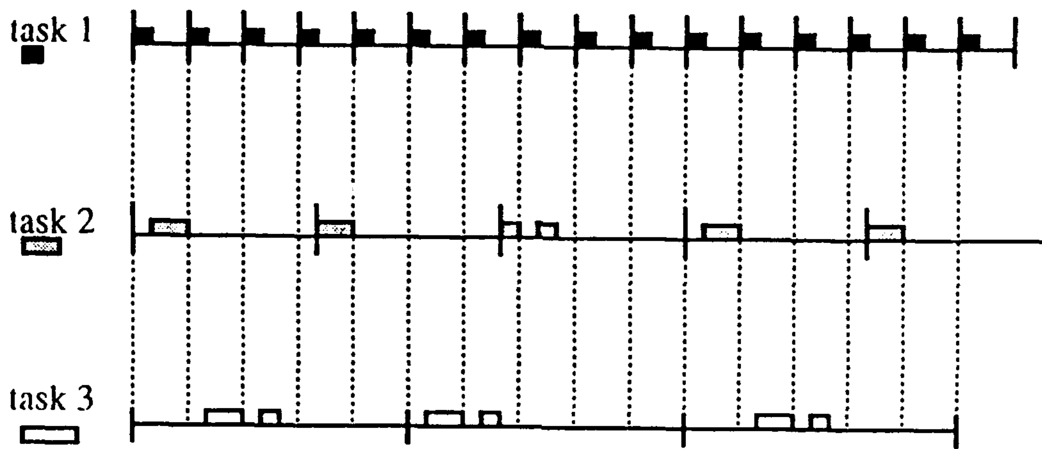


Fig. 1. Example application of the rate monotone algorithm. The inequality of Eqn. (1) is satisfied and no deadlines are missed.

repetition periods (i.e., τ_i and p_i) are indicated on the figures. In Fig. 1, the utilization factors are such that the inequality of Eqn. (1) is satisfied (i.e., $0.7333 \le 0.7797$). Note from the timing diagram of Fig. 1 that all tasks complete their execution before their deadlines. In Fig. 2, the execution times for the three tasks are the same as those in Fig. 1, however, the repetition periods of tasks 2 and 3 are reduced to where the inequality of Eqn. (1) is no longer satisfied (i.e., $0.8833 \not\le 0.7797$). From the timing diagram in Fig. 2, note that no deadlines are missed during the shown time interval. This example shows that the inequality of Eqn. (1) is (only) a sufficient condition for meeting deadlines. In other words, the rate monotone may provide valid schedules even when the inequality is violated. Fig. 3 shows an example where the inequality is violated and the rate monotone algorithm fails to provide a feasible schedule. This demonstrates that, in general, the inequality must be satisfied in order to guarantee no missed deadlines.

$\tau_1 = 1 \quad p_1 = 3$
$\tau_2 = 2 \quad p_2 = 8$
$\tau_3 = 3 \quad p_3 = 10$

$$\sum_{i=1}^{3} \frac{\tau_i}{p_i} \not\leq 3\left(2^{1/3} - 1\right)$$

$$(0.8833 \not\leq 0.7797)$$

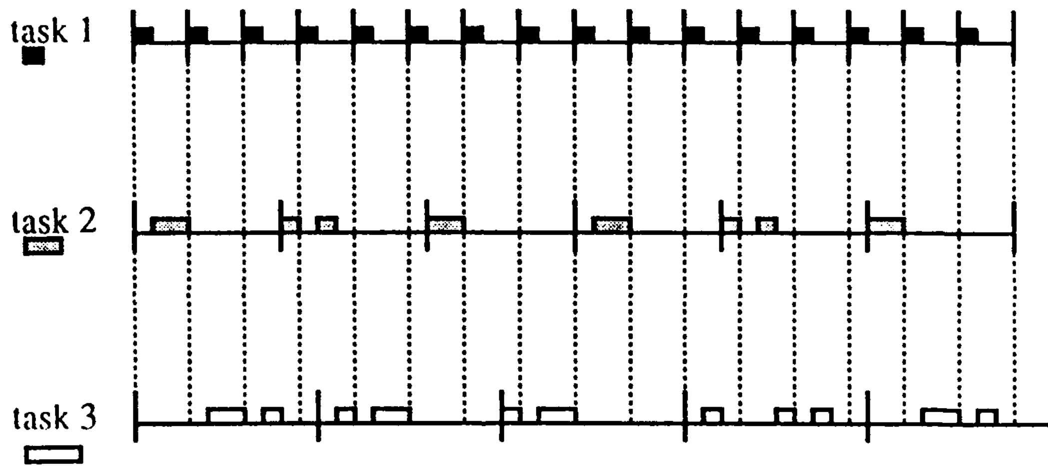


Fig. 2. Example application of the rate monotone algorithm. The inequality of Eqn. (1) is not satisfied and no deadlines are missed. This demonstrates that the inequality is (only) a sufficient condition for no missed deadlines.

$\tau_1 = 1 \quad p_1 = 3$
$\tau_2 = 2 \quad p_2 = 6$
$\tau_3 = 3 \quad p_3 = 10$

$$\sum_{i=1}^{3} \frac{\tau_i}{p_i} \nleq 3\left(2^{1/3} - 1\right)$$

$$(0.9667 \nleq 0.7797)$$

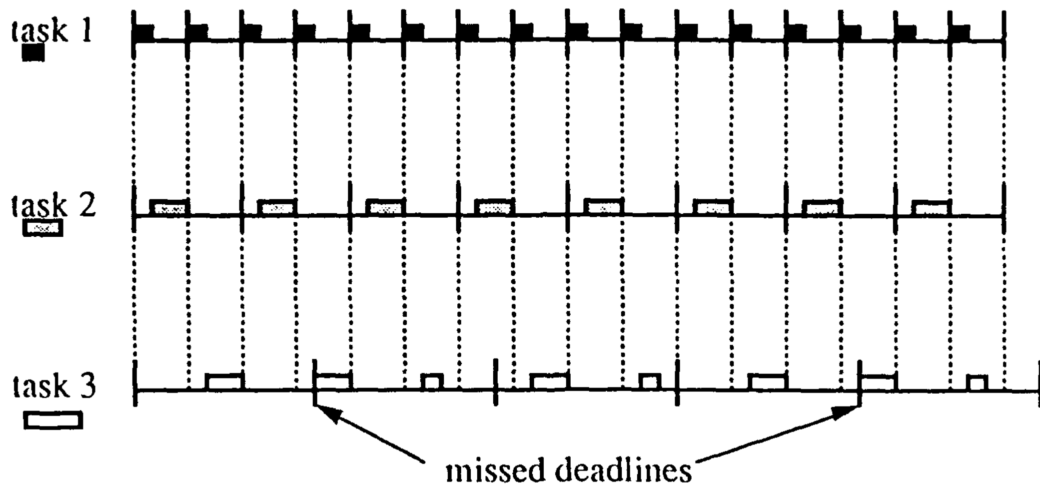


Fig. 3. Example application of the rate monotone algorithm. The inequality of Eqn. (1) is not satisfied and some deadlines are missed. This demonstrates that, in general, the inequality must be satisfied in order to guarantee no missed deadlines.

The problem of how to schedule real-time tasks for multiprocessor systems has also been studied by the real-time computing community. A common approach to the problem of scheduling real-time tasks on multiple processors is to first assign (i.e., map) the tasks onto the processors once and for all, and then schedule the tasks assigned to each processor independently of the tasks assigned to other processors [6]. In order to decrease the likelihood of a missed deadline, it is desirable in practice to assign the tasks so that the processor utilizations are as uniform as possible across all processors. In [3], a greedy heuristic for assigning tasks to processors is analyzed in terms of how its assignments compare to an optimal assignment (where an optimal assignment is one that yields a minimal variation in processor utilizations). It is proven in [3] that the simple heuristic algorithm produces near optimal assignments.

An important assumption made in [3], and one that is typically made for the task assignment problem (within the real-time computing area), is that the cost of interprocessor communication is independent of the assignment. While this assumption is reasonable for some multiprocessor systems designed specifically for real-time applications (which often employ a common bus for interprocessor communication, see for example [19]), it is not generally valid for commercially available massively parallel processing systems. This is because the delay characteristics for the types of interconnection networks often used in commercial systems are sensitive not only to the volume of network traffic but also to the interprocessor communication pattern. Thus, because the interprocessor communication pattern depends on how the tasks are mapped onto the processors, the performance of these networks do generally depend on task mapping.

III. Mapping Tasks onto the Hypercube Architecture

A. The Hypercube Architecture

The hypercube architecture has been a popular choice for interconnecting large numbers of processing elements in parallel processing systems. Some of the attractive features of the hypercube are: a relatively low number of incident links at each processor (node degree = $n = \log_2 N$), a small hop distance between processors (network diameter = $n = \log_2 N$), and a large number of alternate paths between processor pairs. Examples of commercially available MIMD machines that utilize a hypercube interconnection topology include nCUBE's nCUBE 2 and Intel's iPSC2.

An n-dimensional hypercube has two connected processors along each of $\underline{n}$ dimensions for a total of $\underline{N} = 2^n$ processors. By labeling the processors from 0 to $N-1$, the processor-to-processor interconnection pattern is conveniently defined using these processor labels as follows: there is a direct communication link between two processors if and only if the binary representation of their addresses differ in exactly one bit position. For example, in a 3-dimensional hypercube, processor 0 is directly connected (only) to processors 1, 2, and 4. The interconnection pattern for a 3-dimensional hypercube is illustrated in Fig. 4.

B. The Standard Hypercube Embedding Problem

The standard hypercube embedding problem, as typically defined in the parallel processing literature, is to map a given collection of tasks onto the processors of the hypercube topology so that the available communication resources are effectively utilized. Assumed to be given is the intertask communication demands. For general intertask communication patterns, most formulations of the hypercube embedding problem are known to be NP-Hard [7].

A specific objective often used for the hypercube embedding problem is to minimize

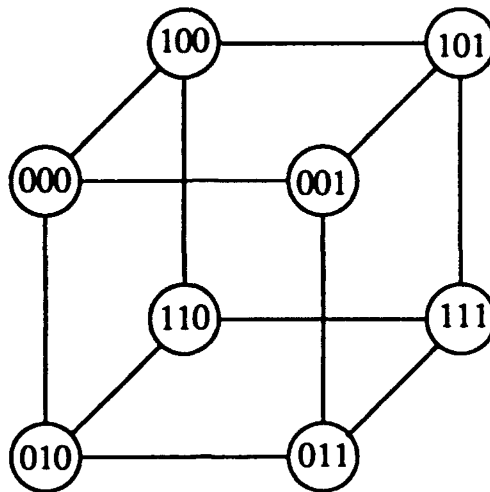


Fig. 4. The network topology of a 3-dimensional hypercube.

the average Hamming distance (i.e., path length) between those pairs of tasks that require communication. For the case of circuit-switched and virtual cut-through routing schemes [10], minimizing the average distance between communicating process pairs reduces the total number of communication links needed to establish all required connections and can therefore potentially reduce the latency caused by contention for common links.

Figs. 5 and 6 show example mappings of eight tasks onto the processors of a eight-node hypercube. The intertask communication demand pattern is the same for both cases and is depicted graphically on the left side of each figure. For instance, the directed link from task T1 to task T5 indicates that task T1 sends a message to task T5. The task-to-processor mapping of Fig. 5 is such that the average distance between communicating tasks is 2.25. In contrast, the task-to-processor mapping of Fig. 6 produces an average distance of 1.00. Also, from the hypercube architectures depicted on the right side of each figure note that some links in Fig. 5 are shared by as many as three paths between communicating tasks while the maximum link utilization for the mapping of Fig. 6 is unity.

Many of the proposed algorithms for solving the hypercube embedding problem as-

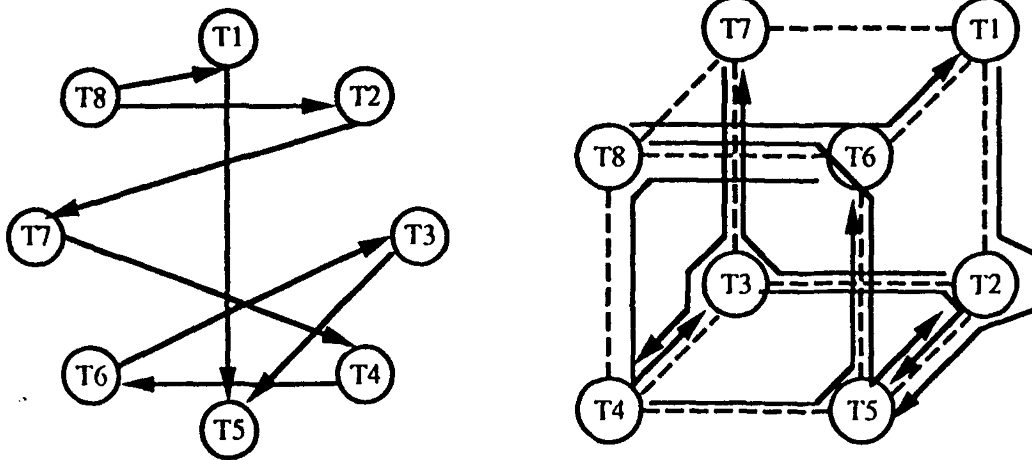


Fig. 5. An example of a task-to-processor mapping onto a hypercube architecture. The average distance between communicating tasks is 2.25. The maximum link utilization is three.

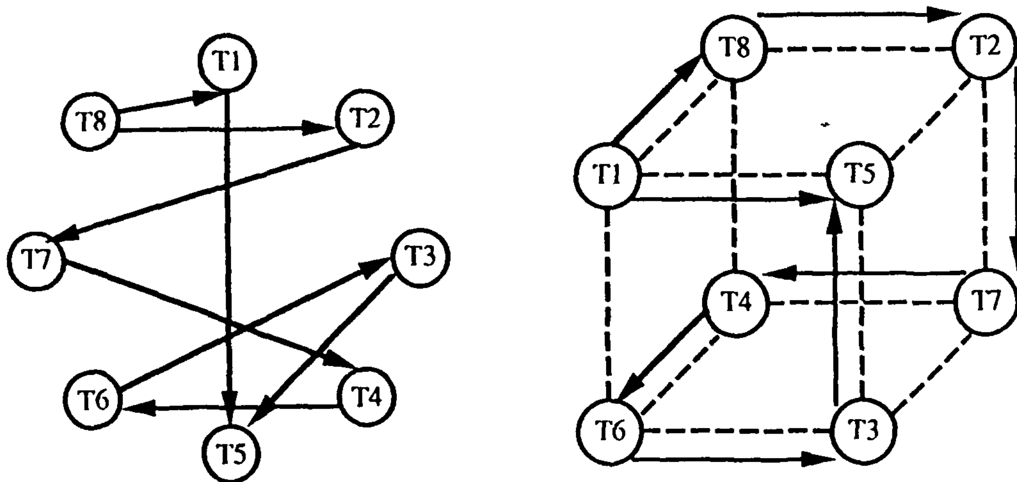


Fig. 6. An example of a task-to-processor mapping onto a hypercube architecture. The average distance between communicating tasks is 1.00. The maximum link utilization is unity.

sume that N or fewer tasks are to be mapped onto the N processors of a hypercube. For a thorough discussion and evaluation of such techniques, refer to [5] and the references therein.

In [1], a nonlinear programming approach is introduced for solving the hypercube embedding problem. The basic idea of the approach is to approximate the discrete space of an n-dimensional hypercube, i.e., $\{z : z \in \{0,1\}^n\}$, with the continuous space of an n-dimensional hypersphere, i.e., $\{x : x \in \Re^n \; \& \; ||x||^2 = 1\}$. The mapping problem is initially solved in the continuous domain by employing the gradient projection technique to a continuously differentiable objective function. The optimal tasks "locations" from the solution of the continuous hypersphere mapping problem are then discretized onto the n-dimensional hypercube. Unlike many past approaches, the technique proposed in [1] can solve, directly, the problem of mapping K tasks onto N processors for the general case where $K > N$.

IV. Mapping Periodic Real-Time Tasks onto the Hypercube

The problem of how to effectively map periodic real-time tasks onto a hypercube architecture is illustrated through an example in this section. As discussed previously in Subsection II-B, when mapping periodic real-time tasks onto multiple processor systems, it is important to achieve balanced utilization of the processors. Also, as discussed in Section III, when mapping tasks onto the hypercube it is desirable to map the tasks so as to minimize contention for the resources of the interconnection network. Thus, for the problem of mapping periodic real-time tasks onto a hypercube architecture, the need to produce uniform processor utilizations should somehow be incorporated in the standard hypercube mapping objective of minimizing network contention.

A. *Possible Techniques for Mapping Real-Time Tasks onto the Hypercube*

In order to illustrate the range of possibilities for mapping real-time tasks onto the hypercube architecture, four candidate techniques are described in this subsection.

Random Modulo Mapper (RMM)

This mapping technique strives to balance the *number of tasks* mapped to each processor (not the total utilization assigned to each processor) by mapping all tasks i that satisfy the equation $i - 1 \bmod N = k$, for some k, onto the same processor. Thus, for the case of $N = 8$, tasks 1, 9, 17 would all be mapped to the same processor, and tasks 2, 10, and 18 would all be mapped to another processor. Obviously, this mapping scheme is oblivious to both the task utilization values and the intertask communication pattern. The RMM is included here to provide a baseline for comparison for the other more sophisticated techniques described below.

Allocate Mapper (AM)

This mapping technique strives to balance the *total utilization mapped to each processor* by employing the greedy allocation algorithm of [3]. Initially, the utilization of processor j, denoted by q_j, is set to zero for each processor j. The allocation scheme starts by sorting the task utilizations (i.e., μ_i's) in descending order. The mapping is defined by selecting the next task ℓ from the sorted list, assigning task ℓ to the least utilized processor j, and updating the utilization at processor j according to $q_j \leftarrow q_j + \mu_\ell$, where μ_ℓ is the task with the ℓth largest utilization. As proven in [3], this allocation scheme results in mappings with a variation in processor utilization that is no more than $\frac{9}{8}$ times that of a mapping with the minimal possible variation in processor utilization. Of course this scheme does not take the intertask communication pattern nor the topology of the interconnection network (in this case the hypercube) into consideration.

Allocate-Hypersphere Mapper (A-HM)

This technique involves first applying the AM technique to cluster the tasks into N "macro-tasks," where N is the assumed number of processors. The associated communication pattern among these N macro-tasks is then used as input to the hypersphere mapper algorithm [1]. Thus, this technique is a two phase approach: the first phase attempts to cluster the tasks into macro-tasks with uniform utilizations and the second phase attempts to map these macro-tasks onto the hypercube architecture so as to minimize contention for the network. Other hypercube embedding techniques besides the hypersphere mapper approach could be applied in the second phase; for other possibilities, refer to the techniques evaluated in [5].

Modified Hypersphere Mapper (MHM)

In contrast to the A-HM approach, which attempts to first divide the tasks into N uniform clusters and then map these clusters onto the hypercube to minimize network contention, the MHM approach attempts to simultaneously satisfy these two objectives. In particular, the objective function of the MHM technique is a modified version of the objective used by hypersphere mapper [1]; it reflects weighted combination of the average distance between communicating tasks and the variance in processor utilization (see [2] for more details). Thus, by minimizing this new objective function, the algorithm searches for mappings that simultaneously balance processor utilization and minimize the effect of network contention.

B. An Example Problem

Fig. 7 shows a portion of a task graph adapted from an air defense system model described in [12]. The air defense system collects sensor data from radar, which is input information to the system. The function of the system is to track all possible attacking vehicles and

command interceptors to protect defended areas. There are 20 tasks shown in the graph numbered from 1 to 20. The directed arcs in the graph denote data dependencies between pairs of tasks. For example, in order for task 4 to complete execution, it must have a data item that is produced by task 0. For simplification of the description here, the data items associated with all intertask data dependencies are assumed to be of the same size.

All of the tasks in Fig. 7 are periodic. The execution times and repetition periods (i.e., τ_i and p_i as defined in Subsection II-B) and the associated utilization factors, $\mu_i = \frac{\tau_i}{p_i}$, are listed in Table I. The tabulated execution times are conservative estimates in that they are based on the assumption that all required intertask communications will require interprocessor communication between two processors of the hypercube. In reality, if two tasks that are linked by a data dependency are both mapped onto the same processor, then no interprocessor communication will be required and thus the effective execution times for the two tasks will be less than the tabulated estimates, because the overhead associated with accessing the interconnection network will not be needed.

The assumed overhead times for an interprocessor communication are 0.2 mseconds for each send and 0.1 mseconds for each receive. So, for example, if task 6 and task 7 are mapped to the same processor, then their actual execution times are reduced from 3 mseconds and 0.8 mseconds down to 2.8 mseconds and 0.7 mseconds, respectively. Note that the tabulated execution times for some of the tasks comprise a relatively large fraction of (assumed) interprocessor communication overhead. For instance, for task 8, 0.9 mseconds of its tabulated 1.5 mseconds of execution time (i.e., 40%) is from assumed interprocessor communication. Thus, there can be significant reduction in actual execution times if the tasks are appropriately clustered onto the processors.

The assumption that the send and receive times (i.e., 0.2 mseconds and 0.1 mseconds, respectively) are distinct is consistent with specifications from existing commercially available hypercube systems, see for example [13]. The implicit assumption that

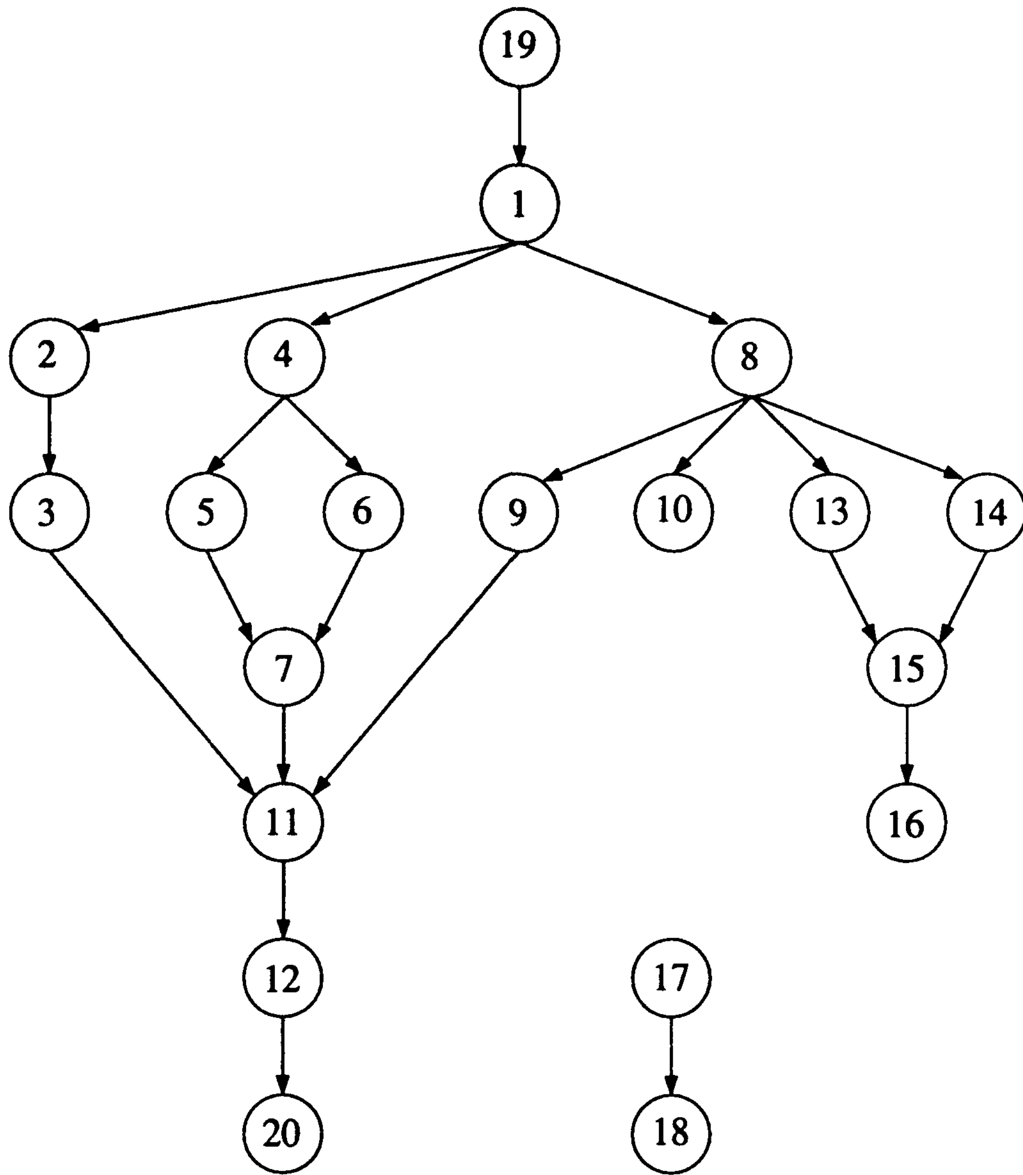


Fig. 7. The example task graph from an air defense system model (adapted from [12]).

TABLE I. TIMING SPECIFICATIONS FOR THE PERIODIC TASKS OF FIG. 7

Task Number i	Repitition Period p_i (mseconds)	Execution Time τ_i (mseconds)	Utilization Factor $\mu_i = \tau_i/p_i$
1	7.10	1.10	0.1549
2	20.00	1.00	0.0500
3	40.00	3.50	0.0875
4	3.20	0.90	0.2813
5	10.00	1.20	0.1200
6	5.50	3.00	0.5455
7	4.10	0.80	0.1951
8	7.00	1.50	0.2143
9	8.60	0.50	0.0581
10	21.00	1.00	0.0476
11	6.70	1.20	0.1791
12	25.00	5.30	0.2120
13	5.00	0.50	0.1000
14	3.50	0.60	0.1714
15	10.10	4.00	0.3960
16	14.30	1.30	0.0909
17	17.00	0.80	0.0471
18	33.00	3.20	0.0970
19	4.90	0.50	0.1020
20	13.60	1.50	0.1103

TABLE II. TASK-TO-PROCESSOR MAPPINGS

Processor	Mapping Technique			
i	RMM	AM	A-HM	MHM
0	2, 10, 18	6	4, 18	1, 2, 9, 17
1	3, 11, 19	15	6	4, 10, 18
2	4, 12, 20	4, 18	15	3, 11, 19
3	1, 9, 17	2, 8, 13, 17	1, 10, 14	12, 20
4	7, 15	9, 12, 19	9, 12, 19	5, 13
5	8, 16	3, 7, 20	3, 7, 20	6
6	6, 14	5, 11, 16	5, 11, 16	7, 15
7	5, 13	1, 10, 14	2, 8, 13, 17	8, 14, 16

these times are fixed and precisely known is an oversimplification of what could be expected in practice. Even though all of the data items being transferred are assumed to be of the same size, the actual times for sending and receiving messages between pairs of processors may increase if network traffic is sufficiently high to cause contention for communication links that are common to paths used to connect distinct pairs of processors.

C. Simulation Results

The four candidate mapping strategies of Subsection IV-A were applied to the problem of mapping the 20 periodic tasks of Fig. 7 onto the eight processors of a 3-dimensional hypercube. The task-to-processor mappings produced by the four mapping techniques are given in Table II. The resulting processor utilization metrics and network metrics for these four mappings are given in Table III and Table IV, respectively.

From Table III, note that the tabulated values for the variance in processor utilization, i.e., var(q), for the AM and A-HM techniques are superior to (i.e., smaller than) the other techniques. As expected, the variance in processor utilization for the RMM mapping is the poorest of the four techniques because RMM does not consider task utilization values

TABLE III. PROCESSOR UTILIZATION METRICS

Processor Utilization Metric	Mapping Technique			
	RMM	AM	A-HM	MHM
q_0	0.2601	0.5455	0.3782	0.2770
q_1	0.1946	0.3960	0.5455	0.4258
q_2	0.3487	0.3782	0.3960	0.3487
q_3	0.5882	0.3628	0.3740	0.3069
q_4	0.2200	0.3722	0.3722	0.2200
q_5	0.7169	0.3929	0.3929	0.5455
q_6	0.5912	0.3900	0.3900	0.5912
q_7	0.3052	0.3740	0.3628	0.4195
min(q)	0.1946	0.3628	0.3628	0.2200
max(q)	0.7169	0.5455	0.5455	0.5912
avg(q)	0.4031	0.4014	0.4014	0.3918
var(q)	0.0445	0.0040	0.0040	0.0194

TABLE IV. NETWORK METRICS

Network Metric	Mapping Technique			
	RMM	AM	A-HM	MHM
Avgerage Hamming Distance	1.41	1.50	1.45	1.18
Total Network Traffic	20	21	21	18

when mapping the tasks onto the processors. The MHM technique does not perform as well as the AM and A-HM techniques in terms of the metric var(q), note, however, that the maximum processor utilization produce by MHM, denoted by max(q), is only slightly higher than that produced by AM and A-HM.

Because the the actual utilization assigned to each processor depends on the amount of required interprocessor communication, note that the average processor utilizations are distinct for three of the mappings. It is clear from the metric avg(q) that the mapping produced by the MHM technique reduces the amount of required iterprocessor communication. For the MHM, four pairs of tasks that require communication are all mapped to the same processor and thus the associated utilizations are reduced because the assumed overhead for accessing the interconnection network are not required. This can be verified by careful examination of the mappings given in Table II relative to the task graph of Fig. 7. For the MHM mapping, the following four pairs of communicating tasks are mapped to common nodes: (1,2), (3,11), (12,20), and (7,13). For the RMM mapping only two pairs of communicating tasks get mapped to a common processor, and for AM and A-HM, only one pair gets mapped to a common processor.

Table IV gives two network metrics for each of the four mapping techniques: average Hamming distance and total network traffic. The average Hamming distance is the average number of links that must be traversed for all pairs of tasks that access the interconnection network. The total network traffic is the number of times the interconnection network is accessed. Note that the MHM technique is superior to the other techniques for both of these network metrics. The RMM and AM techniques do not account for the intertask communication pattern nor the topological structure of the hypercube architecture when mapping tasks to processors. The A-HM technique produces a slightly better average distance measure than the related AM technique. However, the A-HM technique does not match the distance produced by the MHM technique. The reason

TABLE V. SUMMARY OF MAPPING PERFORMANCE METRICS

Aspect of Performance	Mapping Technique			
	RMM	AM	A-HM	MHM
Processor Utilization	poor	excellent	excellent	good
Network Usage	poor	poor	poor	excellent

that the second phase of the A-HM technique (i.e., the application of hypersphere mapper to the macro-task communication graph) does not improve this measure very much from the related AM technique is because the macro-task communication graph produced during the first phase of the technique (from the allocate algorithm) is relatively dense (i.e., each of the eight macro-task requires communication with nearly all other macro-tasks). Thus, all mappings of the eight macro-tasks onto the hypercube have roughly the same average hamming distance. In contrast, the MHM technique does not "lock into" an initial macro-task graph and is thus able to cluster the original 20 tasks in order to simultaneously balance average distance with processor utilization. Table V gives a summary of the mapping results in relative terms.

V. OTHER RESEARCH ISSUES

A. Task Partitioning

As discussed in [3], there are two basic approaches to using multiple processor systems for executing real-time tasks. The first approach is called the partitioning method, which seeks to partition a collection of tasks into groups and assign the groups to distinct processors. The discussion of the mapping problem described in the previous section for hypercube architectures is an application of the partitioning method. The second approach is called the nonpartitioning method, which treats the entire collection of processors as a powerful "virtual processor," and uses this increased processing power to

quickly execute the entire set of tasks sequentially.

The question of how to apply the nonpartitioning approach is closely aligned with some of the research issues and questions found within the parallel processing community (refer to the list of questions listed in Subsection II-A). The following important distinction can be made, however. Within the parallel processing community, the primary reason for exploiting parallelism is to decrease the execution time. Actually, because of the nondeterministic timing behaviors of many commercially available machines, perhaps a more accurate statement of this goal is to decrease the *expected* execution time.

In real-time environments, not only is it desirable to decrease the expected execution time, it is generally also important for the execution time to be highly predictable (i.e., of low variance). As the variability of task execution times increase, it becomes increasingly difficult to effectively schedule tasks to guarantee that all deadlines are always met. Of course, very conservative schedules could be used to account for large execution time variances, but such an approach may defeat the purpose of using a high performance parallel system, because conservative schedules may generally result in unacceptably low processor efficiencies.

B. Operating Systems

In order to effectively apply the partitioning method on an MIMD system for executing real-time tasks, it is necessary for the operating system (usually resident on each processor) to have some real-time task scheduling capability. While recent strides have been made in implementing real-time task scheduling functions within the domain of operating systems for sequential systems, it is not clear if porting this technology into the lean operating system kernels found on most massively parallel MIMD processors is a viable and cost-effective option.

C. I/O

Many commercially available parallel systems are not designed to provide high-throughput bursty I/O. There may be, however, clever ways to overlap computation and communication on such systems so that the I/O subsystems are not a bottleneck for real-time applications.

D. Software Development Process

The software development process for real-time parallel systems is clearly distinct from any current sequential software development process model. The development process for real-time parallel systems is difficult to define because there is currently no sufficient software development model for parallel systems in general. For instance, the problem of how to map real-time tasks onto various classes of architectures is a new problem that must be addressed at some phase of the software development cycle. In order to effectively maintain and upgrade complex parallel software for C3 applications, a meaningful software development process model must be established and adhered to.

VI. CONCLUSIONS

The next generation of C3 systems will likely contain suites of massively parallel processing platforms for executing real-time tasks. The primary goal of this report has been to demonstrate the necessity for a combined effort between researchers in the parallel processing and real-time computing communities.

As noted in [18], there is evidence that some degree of confusion exists concerning how the issues important to real-time computing are influenced by assuming a massively parallel computing platform. An attempt was made in the present report to clear up some of this confusion by giving brief overviews of research issues important to both the parallel processing and real-time computing communities. A specific research problem

involving the mapping of real-time tasks onto a popular type of parallel architecture was introduced to demonstrate the importance of having expertise in both areas.

Representative examples of other more general issues associated with the use of massively parallel processing systems for real-time computing were briefly discussed. These issues represent fertile ground on which significant research can grow.

REFERENCES

[1] J. K. Antonio and R. C. Metzger, "Hypersphere mapper: a nonlinear programming approach to the hypercube embedding problem," *Proc. 7th Int'l Parallel Processing Symposium*, Apr. 1993, pp. 538–547.

[2] J. K. Antonio and R. C. Metzger, "Mapping periodic tasks onto hypercube architectures," Technical Report, School of Electrical Engineering, Purdue University, (under preparation) 1993.

[3] J. A. Bannister and K. S. Trivedi, "Task allocation in fault-tolerant distributed systems," *Acta Informatica*, **20**, Springer-Verlag, Heidelberg, 1983, pp. 261–281.

[4] T. Blank, "The MasPar MP-1 architecture," *Proc. IEEE Compcon*, Feb. 1990, pp. 20–24.

[5] W. K. Chen, M. F. M. Stallman, and E. F. Gehringer, "Hypercube embedding heuristics: an evaluation," *Int'l J. Parallel Programming*, vol. 18, no. 6, 1989, pp. 505–549.

[6] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Trans. Computers*, vol. C-39, no. 9, Sept. 1990, pp. 1156–1174.

[7] G. Cybenko, D. Krumme, and K. Venkataraman, "Fixed hypercube embedding," *Information Processing Letters*, **25**, 1987, pp. 35–39.

[8] M. J. Flynn, "Very high-speed computer systems," *Proceedings of the IEEE*, vol. 54, no. 12, Dec. 1966, pp. 1901–1909.

[9] M. Hewish, "Integrated avionics: the heart of future combat aircraft," *Defense Electronics & Computing*, no. 9, 1992, pp. 1007–1011.

[10] P. Kermani and L. Kleinrock, "A tradeoff study of switching systems in computer communications networks," *IEEE Trans. Computers*, vol. C-29, no. 12, Dec. 1980, pp. 1052–1060.

[11] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, no. 1, Jan. 1973, pp. 41–61.

[12] P.-Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. Computers*, vol. C-31, no. 1, Jan. 1982, pp. 41–47.

[13] nCUBE Corporation, *nCUBE 2 Processor Manual*, Order # 101636, nCUBE Corporation, Dec. 1990.

[14] S. Nelson, "Establishing an MPP guidepost," *Proc. Fourth Symposium on the Frontiers of Massively Parallel Computation*, Oct. 1992, pp. 471–473.

[15] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies, Second Edition*, McGraw-Hill, New York, NY, 1990.

[16] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel processing systems," *Computer*, vol. 25, Feb. 1992, pp. 54–63.

[17] H. J. Siegel, W. G. Nation, and M. D. Allemang, "The organization of the PASM reconfigurable parallel processing system," *Proc. 1990 Parallel Computing Workshop*, Ohio State University, Mar. 1990, pp. 1–12.

[18] J. A. Stankovic, "Misconceptions about real-time computing," *Computer*, Oct. 1988, pp. 10–19.

[19] C. B. Weinstock, "SIFT: system design and implementation," *Proc. Tenth Int'l Symposium on Fault Tolerant Computing*, 1980, pp. 75–77,

[20] J. Xu, and D. L. Parnas, "On satisfying timing constraints in hard real-time systems," *IEEE Trans. Software Eng.*, vol. SE-19, no. 1, Jan. 1993, pp. 70–84.

[21] W. Zhao, K. Ramamrithan, and J. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Trans. Software Eng.*, vol. SE-13, May 1987, pp. 564–577.

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.