Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems

Dar-Tzen Peng, Kang G. Shin, and Tarek F. Abdelzaher

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122

ABSTRACT

We present an optimal solution to the problem of allocating communicating periodic tasks to heterogeneous processing nodes (PNs) in a distributed real-time system. The solution is optimal in the sense of minimizing the maximum normalized task response time, called the *system hazard*, subject to the precedence constraints resulting from intercommunication among the tasks to be allocated. Minimization of the system hazard ensures that the solution algorithm will allocate tasks so as to meet all task deadlines under an optimal schedule, whenever such an allocation exists.

The task system is modeled with a task graph (TG), in which computation and communication modules, communication delays, and intertask precedence constraints are clearly described. Tasks described by this TG are assigned to PNs by using a branch-and-bound (B&B) search algorithm. The algorithm traverses a search tree whose *leaves* correspond to potential solutions to the task allocation problem. We use a bounding method that prunes, in polynomial time, non-leaf vertices that cannot lead to an optimal solution, while ensuring that the search path leading to an optimal solution will never be pruned. For each generated leaf vertex we compute the exact cost using the algorithm developed in [1]. The lowest-cost leaf vertex (one with the least system hazard) represents an optimal task allocation.

Computational experiences and examples are provided to demonstrate the concept, utility, and power of the proposed approach.

Key Words — Branch-and-bound (B&B) algorithm, computation and communication modules, intertask communication, precedence and timing constraints, task invocation and release times, lower-bound cost.

The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-94-1-0229 and the National Science Foundation under grant No. MIP-9203895. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of the funding agencies.

1 Introduction

The workload in a real-time system consists of periodic and aperiodic tasks. Periodic tasks are the "base load" invoked at fixed time intervals while aperiodic tasks are the "transient load" generated in response to environmental stimuli. Periodic servers, like the deferrable server [2], are typically used to handle aperiodic task execution requests. The total execution time of aperiodic tasks is "charged" to the periodically-replenished execution budget of the server, essentially converting the "lumped aperiodics" to periodic equivalents. In hard real-time systems, task execution must be not only logically correct but also completed in time. A pre-runtime analysis is therefore required to guarantee *a priori* that all task deadlines will be met. Moreover, in a distributed real-time system, the ability to meet task deadlines largely depends on the underlying task allocation, and hence, we need a pre-runtime task allocation algorithm that takes into consideration the real-time constraints. Since the end-to-end system response time of distributed applications is affected significantly by intertask communication, one must account for the effect of delays and precedence constraints imposed by intertask communication when task-allocation decisions are made.

In this paper, we deal exclusively with pre-runtime allocation of communicating periodic tasks to processing nodes (PNs) in a distributed real-time system. By "allocation" we mean *assignment* with subsequent *scheduling* considered. The value of our objective function (to be described later), associated with a given allocation, tells whether or not the allocation is *feasible*; that is, whether or not it is possible to schedule tasks under the given assignment such that all of their deadlines and precedence constraints can be met. This is in contrast to conventional methods which deal with either assignment or scheduling of tasks alone, but not both. Our allocation algorithm finds a feasible allocation, if any.

Allocation of aperiodic tasks can usually be treated as a *dynamic load sharing* problem and is beyond the scope of this paper. (See [3–5] for examples of dynamic load sharing in distributed real-time systems.) The three main features of our task allocation problem are:

F1 Tasks communicate with one another during the course of their execution to accomplish a common system goal, thus generating precedence constraints among them. These precedence constraints must be taken into account when deriving an optimal task allocation.

F2 The tasks to be allocated are invoked periodically at fixed time intervals during the mission lifetime.

F3 Tasks are time-critical, meaning that each task must be completed before its deadline, otherwise serious consequences may ensue.

F1 and F2 describe the structure of the task system under consideration, while F3 suggests the objective function for the task allocation problem. F3 requires each task's response time (the time interval between a task's invocation and completion) to be no larger than its relative deadline (the value of a task's deadline relative to its invocation time). The ratio of a task's response time to its relative deadline is called *normalized task response time* [1]. This ratio should not exceed 1 for the task to meet its deadline. Also, in order for all tasks to meet their deadlines, the *maximum* normalized task response time must not exceed 1. The objective function, called the *system hazard*, is therefore defined to be the maximum normalized task response time over all tasks in the system. The algorithm presented in this paper searches for an allocation that minimizes the system hazard (and may be terminated once it finds an allocation whose hazard is less than 1).

The rest of the paper is organized as follows. Section 2 reviews the related work and differentiates our approach from others. Section 3 describes the system model and states the problem. Section 4 gives an overview of the proposed B&B algorithm. Section 5 presents illustrative examples and Section 6 presents computational results, demonstrating the utility and power of the algorithm. The paper concludes with Section 7.

2 Related Work

Task assignment and scheduling problems are studied extensively in both fields of Operations Research and Computer Science [6–14]. For a set of independent periodic tasks, Dhall and Liu [15] and their colleagues developed various assignment algorithms based on the *rate monotonic scheduling* algorithm [16], or *intelligent fixed priority algorithm* [17]. However, if there exist precedence constraints among tasks like our task system, these algorithms cannot be used. Instead, an approach to general task assignment problems must be taken. Depending on the assumptions and the objective functions used, general task assignment problems are formulated differently. However, most prominent methods for task assignment in distributed systems are concerned with minimizing the sum of task processing costs on all assigned processors and interprocessor communications (IPC) costs. Examples of such methods typically include graph-theoretic solutions [18, 19] and integer programming solutions [20] among others [20–23]. Note that minimizing the aggregated processing of, and/or communication cost among, all tasks does not guarantee that individual deadlines will be met. Generally, it is difficult to add real-time constraints when a graph-theoretic approach is used. Integer programming methods, on the other hand, allow for timing constraints. However, these constraints do not account for task queueing and intertask precedence constraints.

Since the problem of assigning tasks subject to precedence constraints is generally NP-hard [24–27], some form of enumerative optimization or approximation using heuristics needs to be developed for this problem. For example, in [28] an enumeration tree of task scheduling is generated and searched using a heuristic algorithm called the CP/MISF (Critical Path/Most Immediate Successors First) and an optimal/approximate algorithm called the DF/IHS (Depth-First/Implicit Heuristic Search) to obtain an approximate minimum schedule length (i.e., makespan) for a set of tasks. Chu and Lan [8] chose to minimize the maximum processor workload for the assignment of tasks in a distributed real-time system. Workload was defined as the sum of IPC and accumulated execution time on each processor. A wait-time-ratio between two assignments was defined in terms of task queueing delays. Precedence relations were used, in conjunction with the wait-time-ratios, to arrive at two heuristic rules for task assignment. Under a slightly different model, Chu and Leung [29] presented an optimal solution to the task assignment problem in the sense of minimizing *average* task response time subject to certain timing constraints. Since constraints are defined in terms of average performance, their models are not suitable for *hard* real-time systems. Shen and Tsai [21], Ma *et al.* [20], and Sinclair [22] derived optimal task assignments to minimize the sum of task execution and communication costs with the branch-and-bound (B&B) [30] method. The computational complexity of this method was also evaluated using simulation in [20, 22]. Minimizing the sum of task execution and communication costs, however, in itself does not guarantee that all task deadlines will be met, since an allocation that causes a short task to miss its deadline may be preferred to the one that doubles the execution time of a significantly longer task, yet meeting all deadlines.

More recent results have been reported in the literature that deals with some or all of the foregoing three features, F1 — F3, of distributed real-time systems. They are more directly applicable to hard real-time systems and have been motivated primarily by the need of contemporary embedded systems whose growing complexity of software and hardware requires an automated resource allocation approach. For example, task allocation algorithms have been reported for process control [31, 32], turbo engine control [33], autonomous robotic systems [34], and avionics [35]. AI-based approaches that utilize application domain knowledge are described in [31, 34, 35]. Solutions to the allocation problem have also been presented for specific hardware topologies such as hypercubes [36], hexagonal architectures [37] and mesh-connected systems [38].

While these application- or topology-specific approaches are efficient in solving the allocation problem for the particular real-time system at hand, those algorithms based on "abstract" task and resource models have the merit of more general applicability. Several such algorithms have recently been reported in the literature. The complexity of the allocation problem usually calls for the use of heuristic solutions. Simulated annealing [39] has been proposed as an optimization heuristic. Different flavors of using simulated annealing in the context of real-time task assignment and scheduling can be found in [40–43]. The quality of the solution found using simulated annealing depends on the selected energy-decay function, and in general, is not guaranteed to be optimal.

Other heuristic solutions to task allocation in real-time systems include graph-theoretical, communication-oriented and schedulability oriented methods. For example, [44] considers an abstract problem where a given task graph is invoked periodically under an end-to-end deadline. A task allocation and message schedule are computed such that the end-to-end deadline is satisfied for each invocation. In [45–47] efficient methods are considered for allocating periodic tasks where different tasks may have different deadlines. To reduce the size of the problem, tasks are *pre-clustered* before allocation begins. Task clusters (instead of individual tasks) are then assigned to individual processors. Graph-based heuristics, which attempt to minimize interprocessor communication, are used for task assignment in [45, 46]. In contrast, the approach in [47] searches the space of all (cluster) assignments and schedules until either a feasible solution is found or the space of all solutions have been searched exhaustively. The search is directed by a heuristic that explores the most "likely" parts of the search space first and is usually efficient in finding a feasible solution, if any. However, since tasks are pre-clustered without considering their timing constraints, optimality can be lost because of clustering. Moreover, the approach was concerned with homogeneous systems and non-preemptive scheduling only.

Optimal solutions for some class of real-time task allocation problems have been reported in the literature. For example, [48] describes an optimal branch and bound (B&B) algorithm for task assignment and scheduling on multiprocessors subject to precedence and exclusion constraints. In [49] a task assignment and scheduling algorithm is presented to optimally minimize the *total execution time* (TET) of an arbitrary task graph in a distributed real-time system. It employs a stochastic optimization phase to find a solution with "good" TET. The computed TET then serves as an upper bound to restrict the search space of a subsequent mixed integer-linear programming optimization phase that finds the optimal allocation. The algorithm is applicable to systems with a single end-to-end deadline. In systems where different tasks have different deadlines minimizing TET does not necessarily lead to a feasible schedule.

Fault-tolerance requirements have also been considered in task allocation problems. A *k-Timely-Fault-Tolerant* problem is solved in [50] where an assignment and schedule are found for replicated tasks such that all deadlines are met in the presence of up to k processor failures. The solution in [50] is concerned with independent tasks only. In [47, 51] replicated tasks with precedence constraints are considered. While [47] uses a deterministic task execution model, the algorithm of [51] solves a probabilistic model trying to maximize the probability of missing task deadlines. Although it is optimal in minimizing the performance measure, the algorithm in [51] requires information about the task model which may not be available (for example, branching probabilities of individual branch statements). Instead, we use a deterministic approach where worst-case execution times are sufficient to represent task load.

We present an optimal algorithm that allocates communicating periodic tasks in heterogeneous distributed real-time systems. The algorithm is optimal *both* in the sense that (i) a feasible solution (i.e., a solution where all deadlines are met) is found whenever one exists *and* that (ii) the resulting solution minimizes maximum normalized task response time, or the system hazard. We show in Section 3 that (ii) implies (i). Note however, that (ii) is stronger since it specifies the behavior of the algorithm even when no feasible solution exists. If the problem is unsolvable, the algorithm returns the closest-to-feasible solution, in the sense of minimizing the objective function. In our task model, replicated tasks can be expressed as separate tasks with *allocation constraints*. In general, allocation constraints dictate if particular tasks cannot be co-allocated (e.g., replicas of the same computation), must specify which tasks must be co-allocated, must be allocated to particular processors, or cannot be allocated to particular processors. To our knowledge, the algorithm presented in this paper is the first optimal task allocation result for heterogeneous distributed systems and precedence constrained communicating periodic tasks.

3 Task System and Problem Statement

In order to meet the deadlines of communicating tasks in a distributed real-time system, it is necessary to accurately describe the computational load imposed by each task, the effects of intertask communication and the ability of each PN to execute tasks. In what follows, we first introduce the task model and important definitions to be used, then formulate the allocation problem.

3.1 Task System Model

Let $T = \{T_i : i = 1, 2, ..., m\}$ be the set of $m \geq 2$ periodic tasks to be allocated among a set of $n \geq 2$ processing nodes, $N = \{N_q : q = 1, 2, ..., n\}$ of the system. Let p_i be the period of task $T_i \in T$, and L be the least common multiple of all p_i's. The interval $I = [0, L)$ is called the *planning cycle* of T. For this task set, it suffices to analyze the behavior of the system only in one planning cycle, since it will repeat itself for all subsequent planning cycles. A set, AC, of allocation constraints may be imposed on tasks. The most common constraints are (i) T_i and T_j must be allocated to the *same* processor, (ii) T_i and T_j must be allocated to *different* processors (e.g., replicated tasks), or (iii) T_i must be allocated to $N_q \in P \subset N$.

Let T_{iv} be the v-th invocation of task T_i within a planning cycle (i.e., $0 \leq v \leq (L-1)/p_i$). The *release time* of T_{iv} is the earliest time it may start execution, which is vp_i. In the task model considered here, the deadline of T_{iv} must be on or before the release time of the next invocation, which is $(v+1)p_i$. Each invocation T_{iv} of task $T_i \in T$ consists of one or more *task modules*, which are the smallest objects to be scheduled. Precedence constraints may exist between modules of the same or different tasks. A precedence constraint, $M_j \prec M_k$, between two modules M_j and M_k means that M_j *precedes* M_k, i.e., M_k cannot start execution before M_j completes. Let PR be the set of all precedence constraints defined in the system.

A task module M_j can be one of two types: *computation module* or *communication module*. The execution time of a computation module depends on processor speed. Communication modules do communication-related processing. In the current model each communication module has a single communication partner, which is a communication module in a different task. The execution time of communication modules depends both on processor speed and on whether or not the communication partner is remote (i.e., assigned to a different PN). Let e_{jq} denote the execution time of a computation module M_j on processor N_q. The execution time of a communication module M_j on N_q is denoted by $e_{jq_{remote}}$ if the partner is remote, and $e_{jq_{local}}$ if the partner is local, where $e_{jq_{local}} \leq e_{jq_{remote}}$. For example, the execution of an RPC to some server can be modeled by a communication module. If the RPC is remote, the execution time of the module is larger than when the RPC is local due to the additional overhead of executing a network communication protocol. Note that the difference in module execution time is due to processing at the end host. Network delays are considered separately since they do not consume host's processing power.

Let d_{ij} be the network delay between two communicating modules M_i and M_j allocated to different PNs. It includes network propagation delays, link/packet contention delays as well as the delays incurred by communication processors, if any, in processing the message *en route*. (For example, dedicated network processors may be used for routing and/or switching at the source and intermediate nodes.) Of these, contention delays are usually dominant and depend on the underlying routing and switching strategies. For example, in a *store-and-forward* switching network where an intermediate node must receive a complete packet before forwarding it to the next node, the hop count between the PNs on which modules M_i and M_j reside, denoted by n_{ij}, as well as message length, l_{ij}, generally affect the delay most. In a wormwhole or circuit-switched network, on the other hand, where a dedicated circuit is set up between source and destination over which data is pipelined, the delay is affected mostly by the length, l_{ij}, of the communicated message. In general, for some hybrid switching scheme like the one in [52], the delay may be approximated by $d_{ij} = an_{ij} + bl_{ij} + cn_{ij}l_{ij}$, where a, b and c are constants for the particular network. Note that the above expression is merely an approximation. If a particular communication paradigm (e.g., *real-time channels* [53]) is known then d_{ij} may be expressed precisely. Our allocation algorithm does not make assumptions as to how d_{ij} is computed. Also, note that module execution times and communication delays may be interpreted as worst-case estimates when exact values cannot be obtained.

To describe the task system within a planning cycle, an acyclic directed task graph (TG) is used, where modules are represented by boxes labeled with module numbers, and precedence constraints are represented by arcs. The graph contains all invocations of all tasks in one planning cycle. Figure 1 is an example task graph for a three task system. Communication modules in the graph are shaded to distinguish them from computation modules. Communication partners are connected by bold arrows representing the delay d_{ij} and the direction of communication. Note that the periods of T_1 and T_2 are 40 while the period of T_3 is 20. Thus, T_3 has two invocations in the task graph. Also note that we distinguish modules belonging to different invocations of the same task, because we view a module as an execution instance of a certain chunck of code, rather than the chunk of code itself. We call a module which may execute last in a task invocation (such as M_4, M_5, M_{16}, M_{20}, and M_{24}) is called a *completing module*. Such modules are of special interest since the completion time of a task invocation, used to compute the system hazard, is always equal to the completion time of a completing module.

3.2 Problem Statement

Consider the task set T. Let r_{iv}, and d_{iv} be the release time and absolute deadline of the v-th invocation, T_{iv}, respectively. Let c_{iv} be the invocation's completion time under some allocation δ. Define the *normalized task response time*, $\overline{c}_{iv}$, of a task invocation T_{iv} as

$$\overline{c}_{iv} = \frac{c_{iv} - r_{iv}}{d_{iv} - r_{iv}} \tag{3.1}$$

Notice that $\overline{c}_{iv}$ depends on the allocation δ. The *system hazard*, Θ^{δ} for an allocation δ is defined as:

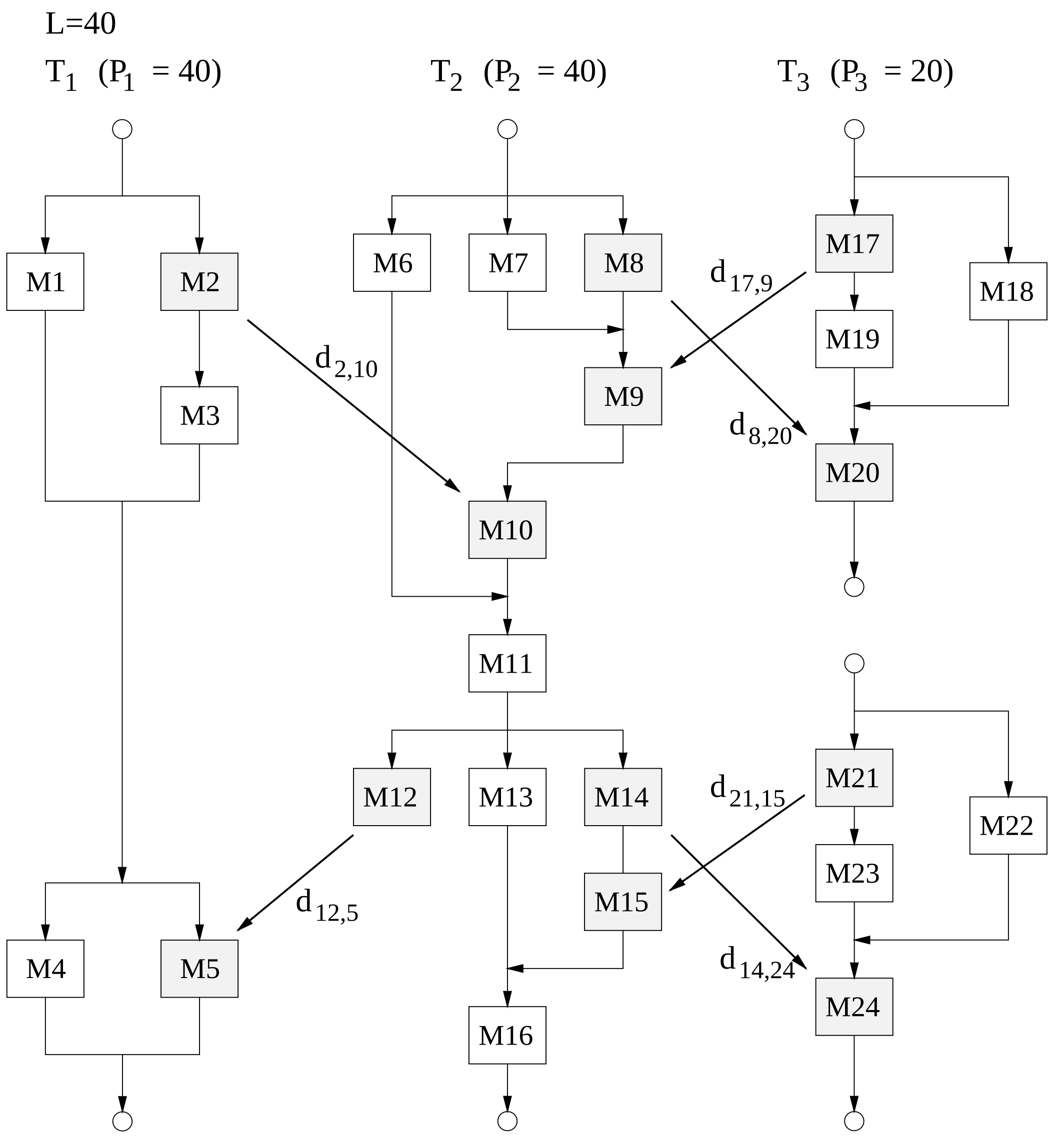

Figure 1: An Example Task Graph.

$$\Theta^{\delta} = \max_{T_i \in T} \overline{c}_{iv} \tag{3.2}$$

In other words, Θ^{δ} is the maximum normalized response time, $\overline{c}_{iv}$, over all invocations of all tasks in T (in a planning cycle). We want to find an optimal task allocation δ^* that minimizes the system hazard, i.e., $\Theta^{\delta^*} = \min_{\delta} \Theta^{\delta}$. Note that $\overline{c}_{iv}$ (and thus Θ^{δ}) depends on how the tasks are assigned under δ *and* how they are scheduled on each PN. As mentioned in Section 1, an allocation δ is feasible if a task assignment and schedule is found such that all task invocations meet their deadlines (i.e., $\forall T_{iv} : c_{iv} \leq d_{iv}$). From Equations (3.1) and (3.2), this means $\Theta^{\delta} \leq 1$.

Minimizing maximum task lateness (i.e., $\max(c_{iv} - d_{iv})$) comes close to the system hazard. Whenever an allocation algorithm γ that minimizes maximum task lateness finds a feasible solution, so does our algorithm does. This is because a feasible solution to task lateness minimization (i.e., one in which all deadlines are met) implies the existence of a solution to the problem of minimizing the system hazard for which $\Theta^{\delta} \leq 1$. This implies that the optimal solution to the system hazard minimization has the property $\Theta^{\delta^*} \leq \Theta^{\delta} \leq 1$, meaning that it too is feasible. The difference between the two solutions would be that in system hazard minimization, task laxity is more likely to be distributed in proportion to task timing constraints. In [1, 54] it is argued that this results in a more "robust" schedule. The argument, however, is a rather subjective. Generally, the choice between the two objective functions may depend on the application at hand. In cases where the only requirement is to find a feasible allocation whenever one exists, both measures are equally acceptable.

4 The Task Allocation Algorithm

The algorithm proposed in this paper employs a branch-and-bound (B&B) technique to solve the allocation problem. Section 4.1 presents a general overview of the proposed algorithm. A detailed description of the proposed algorithm is given in Section 4.2.

4.1 Overview of the Main Algorithm

Our goal is to find a task allocation that minimizes the system hazard. The algorithm considers m tasks, one at a time, in the task set T. Without loss of generality, we can number tasks in the order they are considered for allocation. Thus, if $i < j$ then T_i is allocated prior to T_j. Looking for an allocation which minimizes the system hazard may be viewed as traversing a search tree. Each vertex in the tree corresponds to a partial/complete allocation. Hence, we will occasionally use terms *vertex* and *allocation* interchangeably. The root of the search tree is the null allocation in which no tasks have been assigned. It is expanded by considering all possible assignments of T_1. Each subsequent level in the tree corresponds to assigning the next numbered task. Thus, a vertex at level k in the tree represents an assignment of tasks $\{T_1, ..., T_k\}$. Such a vertex is expanded by considering all possible assignments of T_{k+1} to a PN. Since there are n PNs in the system, each vertex has exactly n children representing all possible ways of assigning the next task (most of which will be pruned in the search process). Since there are m tasks in the system, vertices at level m are the leaves of the tree, corresponding to *complete allocations*, i.e., allocations of all m tasks in T. These are the possible solutions to the task allocation problem. The set of all leaves represents all possible solutions. For example, Figure 2 gives the search tree and all solutions for a system of three tasks and two PNs, N_1 and N_2. Our goal is to find an optimal solution while generating as few vertices in the search tree as possible.

In general, a B&B search that minimizes a performance measure Θ starts at the root of the search tree and computes a *lower bound* $\Theta_{lb}(x)$ of the performance measure (also called the *vertex cost*) at

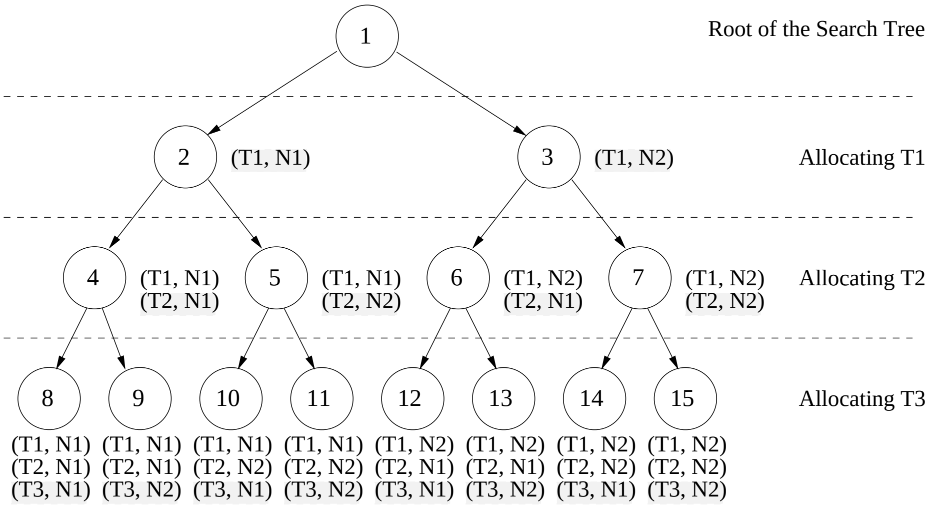

Figure 2: The overall search tree for assigning three tasks on two PNs

each generated vertex x. The bound represents the smallest value of the performance measure that a solution descending from this vertex might have. For example, if in Figure 2 leaf vertices 8, 9, 10 and 11 have system hazards of 0.6, 0.7, 1.2, and 0.4, respectively, then a valid lower bound for, say, vertex 2 is 0.4 or less, and a valid lower bound for, say, vertex 4 is 0.6 or less. Furthermore, 0.4 and 0.6 are called the *exact* lower bounds for vertices 2 and 4, respectively. Note that if the lower bound of the system hazard at some vertex is more than 1, then there are no feasible solutions descending from that vertex. The lower bound is used to prune vertices. In general, if Θ_{lb} at a vertex is higher than the value of performance measure Θ of some already found solution, then the vertex may be pruned because none of its descendants improve on the solution found. Note that in order for pruning to work for the allocation problem defined is Section 3 we must be able to do the following:

- Compute the value of performance measure Θ for a vertex representing a solution. That is, we need a method to compute the system hazard for leaf vertices (i.e., those representing complete allocations). Computing the system hazard for a given task assignment requires an optimal schedule to be generated for that assignment. For this purpose, we use the optimal scheduling algorithm described in [1].

- Compute a good lower bound Θ_{lb} of the system hazard for a vertex. We call it the *cost* of the vertex. Note that at leaf vertices the (exact) lower bound is equal to the actual value of the performance measure computed using the algorithm in [1]. The remaining problem is to compute the cost of non-leaf vertices.

In the rest of this paper, we denote the cost of vertex x by $\Theta(x)$, where at leaf vertices $\Theta(x)$ is the exact system hazard computed using the algorithm in [1], while at a non-leaf vertex it is the lower bound

$\Theta_{lb}(x)$ computed for the vertex. An optimal solution is a leaf vertex with minimum cost.

Our B&B algorithm maintains a set of *active* vertices, which is the subset of all vertices searched that is considered to contain, or lead to, an optimal solution. Initially, this set contains only the root vertex whose cost may be set to zero (a trivial lower bound on the system hazard). The algorithm proceeds by alternating branching and bounding operations. Branching expands the minimum-cost vertex in the active set by generating its children, i.e., adding them to the active set and removing their parent. Bounding evaluates the cost of the newly-added vertices to the active set. The algorithm also keeps track of the minimum leaf vertex cost obtained so far, Θ_{min}. Initially, Θ_{min} is set to infinity. Whenever a better (lower) value for Θ_{min} is found, all vertices with higher costs than Θ_{min} are pruned (i.e., removed from the active set). In the case where two or more leaf vertices have the same cost, only one of them is retained arbitrarily. The algorithm terminates when the active set contains exactly one leaf vertex. It has been proven in [30] that there always exists a leaf vertex surviving the above search process, which is the optimal solution to the original problem.[1]

Figure 3 gives the algorithm's pseudo code. It completely specifies our algorithm except for the function used to compute vertex cost on line 15. This function and the insights behind it are described in the following subsection.

4.2 Computing Vertex Cost

The B&B algorithm needs an efficient bounding process such that an optimal solution can be found in reasonable time. An efficient bounding process tries to prune a generated vertex, x, as early as possible by computing a lower bound on the system hazard associated with the vertex. The tighter (i.e., higher) the lower bound, the more likely the vertex will be pruned at an early stage. We describe a method for computing the lower bound at non-leaf vertices in polynomial time. As mentioned in Section 4.1 at leaf vertices the vertex cost is equal to the system hazard computed as described in [1]. The lower bound, $\Theta_{lb}(x)$, of the system hazard for some non-leaf vertex x is computed in three steps:

Step 1: Compute the minimum computational load imposed on each processor by tasks already assigned to PNs at search vertex x.

Step 2: Estimate the minimum additional load to be imposed on each PN due to those tasks not yet assigned at x.

Step 3: Schedule the combined load at each PN and compute the system hazard. We ensure that the system hazard of the resulting schedule is a lower bound on the system hazard of any leaf vertex descending from x, i.e., it represents $\Theta(x) = \Theta_{lb}(x)$.

We summarize below the measures taken to ensure that the computed value of $\Theta(x)$ is indeed a true lower bound on the system hazard achievable at any of x's descendants.

[1]This is assuming there exists a solution that satisfies all allocation constraints in set AC.

```
let active set A = {Root}
let vertex cost Θ(Root) = 0
let best solution cost, Θ_min = ∞

while true do
    let V_best = minimum cost vertex in A
    if V_best is a leaf vertex then
        prune all vertices V ∈ A except V_best
        return V_best as optimal solution
    else
        generate (task assignments of) all children of V_best
        remove V_best from active set A
        for each child x of V_best do
            if assignment constraints in set AC are not satisfied then prune x
            else
                compute vertex cost Θ(x)
                add x to active set A
                if x is a leaf vertex then
                    if Θ(x) < Θ_min then
                        Θ_min = Θ(x)
                        prune all vertices V ∈ A for which V ≠ x and Θ(V) ≥ Θ_min
                    else prune x
                end if
            end if
        end for
    end if
end while
```

Figure 3: The branch and bound algorithm.

First, as shown in the subsequent sections, the task/module execution times computed in Step 1 and Step 2 are the minimum possible task/module execution times given a partial allocation. Thus, under any complete allocation resulting from the partial allocation at vertex x, the actual task set assigned to each PN will impose a greater load than the one computed in Steps 1 and 2. Second, for the purpose of computing a task schedule for some node N_q, in Step 3, we assume that all task invocations on other PNs with modules which precede a module on N_q start exactly at their release times (i.e., start as early as possible). Third, the scheduling algorithm employed in Step 3 on each PN is locally-optimal with respect to the *node hazard* which is defined as the maximum normalized response time among all tasks assigned to a given node. The node hazard (and therefore the system hazard) computed for vertex x subject to the above assumptions will therefore be smaller than, or equal to, the one computed for x

under a schedule which considers all precedence constraints for the same task set. Thus, it is indeed a true lower bound on the system hazard achievable at any descendent of x.

In what follows we give the details of computing the cost of a non-terminal vertex x. Section 4.2.1 and Section 4.2.2 describe how the load of each PN is computed in Step 1 and Step 2, resectively. The load is represented as a set of jobs, each with a known release time, execution time, deadline, and possibly precedence constraints. Section 4.2.3 shows how these jobs are scheduled in Step 3 to compute $\Theta_{lb}(x)$.

4.2.1 Load of Allocated Tasks

Consider some non-leaf search vertex x for which vertex cost $\Theta(x)$ needs to be computed. Vertex x corresponds to a partial assignment where some tasks have already been allocated to PNs. The first step in computing $\Theta(x)$ is to find the minimum load imposed by these tasks on different PNs in any complete allocation descending from x. This is done by evaluating the minimum execution time for each module of an allocated task, as well as evaluating the minimum communication delays. The following rules are used to compute the minimum load:

- Each computation module M_j allocated to N_q contributes an execution time $e_j = e_{jq}$ on N_q, where e_{jq} is as defined in Section 3.
- Each communication module M_j allocated to node N_q contributes an execution time $e_j = e_{jq_{remote}}$ if its communication partner is remote in the current assignment, and contributes $e_j = e_{jq_{local}}$ otherwise (i.e., if its communication partner is either local or has not been allocated yet).
- A delay d_{ij}, computed as described in Section 3, is inserted between any pair of communicating modules allocated to different PNs. No delay is inserted between communication partners allocated to the same PN, or where one of the partners has not yet been allocated.

For the purpose of scheduling in Step 3, each module M_j of a task invocation (or job) T_{iv} allocated to N_q is viewed as a job on N_q with execution time e_j, deadline d_{iv}, and the precedence constraints defined for M_j in the original precedence constraint set, PR. Only precedence constraints between *allocated* modules are considered (i.e., retained as precedence constraints between the corresponding jobs). As described in the next section, we ignore the precedence constraints with modules that have not yet been allocated. Finally, since M_j cannot start execution before its predecessors, the release time, r_j, of the corresponding job is set to the larger of the release time of T_{iv} and the maximum release time of any task invocation containing a predecessor of M_j. This is expressed in Equation (4.1).

$$r_j = \max\{r_{iv}, \max\{r_{uw} | \exists M_k \in T_{uw} : M_k \prec M_j\}\} \tag{4.1}$$

4.2.2 Load of Unallocated Tasks

Consider the tasks that have not been assigned to PNs at the search vertex x. The second step in computing vertex cost is to estimate the minimum load such tasks will contribute to each PN at any complete allocation descending from vertex x. By scheduling this load in addition to allocated tasks (as will be described in Section 4.2.3), a tight lower bound of the system hazard can be computed for vertex x.

Consider an invocation T_{iv} of some unassigned task T_i. Let $\rho_{q,T_{iv}}$ be the minimum additional load (execution time) to be imposed on node N_q due to T_{iv} with deadline d_{iv}. This load depends on whether or not T_i will be assigned to N_q.

- If T_i is eventually assigned to N_q, T_{iv} will increase the node's load by the sum of its modules' processing times, $\sum_{M_j \in T_{iv}} e_{jq}$. All of that load will have to be executed by deadline d_{iv}.
- If T_i is eventually assigned elsewhere then each module M_j, assigned to N_q, that communicates with T_{iv} will have a longer processing time $e_{jq_{remote}}$ than $e_{jq_{local}}$ assumed by default in Step 1. Of these only the modules that *precede* their communication partners in T_{iv} are necessary for T_{iv} to complete by deadline d_{iv}. Let $\psi_{q,T_{iv}}$ be the set of all such (preceding) communication modules on N_q. Thus, the load increment on N_q will be $\sum_{\psi_{q,T_{iv}}}(e_{jq_{remote}} - e_{jq_{local}})$.

To compute a lower bound on load increment on N_q due to T_{iv}, and since we do not know where T_i will be allocated, we set $\rho_{q,T_{iv}}$ to the smaller of the two load increments mentioned above. The expression for $\rho_{q,T_{iv}}$ is given as:

$$\rho_{q,T_{iv}} = min\left(\sum_{M_j \in T_{iv}} e_{jq}, \sum_{\psi_{q,T_{iv}}}(e_{jq_{remote}} - e_{jq_{local}})\right) \tag{4.2}$$

The above computed load is considered as a lump sum represented by a single job on N_q of execution time $\rho_{q,T_{iv}}$, and deadline d_{iv}. Since $\rho_{q,T_{iv}}$ is, in fact, obtained by adding up execution times (or parts thereof) of modules from different tasks which generally have different precedence constraints, it is not straightforward to compute the resultant "aggregate" precedence constraints on $\rho_{q,T_{iv}}$. To simplify the problem (since we are looking only for a lower bound), precedence constraints on $\rho_{q,T_{iv}}$ are ignored. By neglecting precedence constraints, we assume that the job representing $\rho_{q,T_{iv}}$ can be released as early as possible. Its release time is given as the earliest release time of any task invocation containing a module that contributes to $\rho_{q,T_{iv}}$ as:

$$r_j = \min\{r_{iv}, \min\{r_{uw} | \exists M_k \in T_{uw} : M_k \in \psi_{q,T_{iv}}\}\} \tag{4.3}$$

The release time thus computed reflects the fact that the execution time $\rho_{q,T_{iv}}$ belongs to either the unallocated task invocation T_{iv}, or its communication partners on N_q, expressed by the set $\psi_{q,T_{iv}}$. Since

	A job represents an allocated module $M_j \in T_{iv}$	A job represents an unallocated invocation T_{iv}
r_j	$\max\{r_{iv}, \max\{r_{uw} \mid \exists M_k \in T_{uw} : M_k \prec M_j\}\}$	$\min\{r_{iv}, \min\{r_{uw} \mid \exists M_k \in T_{uw} : M_k \in \psi_{q,T_{iv}}\}\}$
e_j	e_{jq} *or* $e_{jq_{remote}}$ *or* $e_{jq_{local}}$	$\rho_{q,T_{iv}}$
d_j	d_{iv}	d_{iv}
r_{0_j}	r_{iv}	r_{iv}

Table 1: Job parameters on node N_q.

we do not know *a priori* which alternative should be the case, the minimum of the two corresponding release times is taken. This ensures that the system hazard computed from the schedules in Step 3 is a lower bound. Furthermore, minimization over release times of predecessors is performed because different parts of the "lumped" execution time $\rho_{q,T_{iv}}$ are released at different times. Since we deal only with the lumped sum, we do not state which parts are released when. We thus have to assume that all parts are released at the earliest possible time, to guarantee that we always get a true lower bound of the system hazard. Note that unlike the case in Step 1, where each allocated module is translated into a total of *one* job on the processor it is assigned to, unallocated task invocations are translated each into a set of n jobs, one for each PN. Some of these jobs might have zero execution time in which case they are neglected. Section 4.2.3 considers how to schedule jobs on each PN.

4.2.3 Job Scheduling

At search vertex x, let $J = \{J_1, J_2, ..., J_p\}$ be the set of all jobs, with nonzero execution times, generated in Steps 1 and 2. Let $JQ \subset J$ be the subset of jobs assigned to some processing node N_q. As described earlier, each job J_k represents either a module of some task invocation allocated to N_q (see Section 4.2.1) or the minimum aggregate load on N_q by some unallocated task invocation (see Section 4.2.2) at search vertex x. Let r_k, e_k, and d_k denote the release time, execution time and deadline of job J_k as computed in Section 4.2.1 and Section 4.2.2. For notational convenience let r_{0_k} be the release time of the task invocation who/whose module is being represented by J_k. Table 1 summarizes how job parameters have been computed.

A lower bound of the node hazard is computed for each node N_q by applying the optimal uniprocessor scheduling algorithm *Algorithm A* of [55] to the set of jobs, JQ. The algorithm is applied for each processor independently. For completeness, Algorithm A of complexity $O(|JQ|^2)$ is briefly described below, where $|JQ|$ is the number of jobs to be scheduled. The algorithm is optimal with respect to any *regular measure*, where a regular measure is any cost function which increases monotonically with job completion time. In this case we are interested in minimizing the system hazard. (Note that the system hazard is a regular measure.) The following steps show how a set of jobs with arbitrary release times and precedence constraints is to be scheduled on a single machine so as to minimize such a regular measure.

SA1 For the purpose of scheduling, modify job release times, where possible, to meet the precedence constraints among the jobs and then arrange the jobs in nondecreasing order of their modified release times to create a set of disjoint blocks of jobs. Let r'_k denote the modified release time of job J_k of original release time r_k. For example, if jobs J_1, J_2 and J_3 are released at $t = 0, 2, 15$, respectively, J_1 precedes J_2 which in turn precedes J_3, and 5 units of time are required to complete each of J_1 and J_2, then job J_2's release time is modified to $t = 5$ (i.e., $r'_2 = 5$), J_3's is kept at $t = 15$ and two blocks of jobs $\{J_1, J_2\}$, which occupies interval $[0, 10)$, and $\{J_3\}$, which occupies interval $[15, 20)$, will be created.

SA2 Consider a block B with block completion time $t(B)$. Let B' be the set of jobs in B which do not precede any other jobs in B. Select a job J_l such that $f_l(t(B))$ is the minimum among all jobs in B', where $f_l(t)$ is the nondecreasing cost function of job J_l if it is completed at t (in this case $f_l(t)$ is the job's normalized response time). This implies that J_l be the last job to be completed in B.

SA3 Create subblocks of jobs in the set $B - \{J_l\}$ by arranging the jobs in nondecreasing order of modified release times, r'_k, as in SA1. The time interval(s) allotted to J_l is then the difference between the interval of B and the interval(s) allotted to these subblocks.

SA4 For each subblock, repeat SA2 and SA3 until time slot(s) is(are) allotted to every job.

To compute vertex cost we simply substitute the completion times of task invocations from the obtained schedule into Equations (3.1) and (3.2). Let a job J_k be called a *completing job* if it corresponds to either an allocated completing module or an unallocated task invocation. Thus, the completion time, c_k, of such a job is the completion time of a task invocation in the original task graph. Let the set $JC \in JQ$ be the set of all completing jobs in JQ. The lower bound, $\Theta_{lb}(x)$, of the system hazard, associated with vertex x, is computed from:

$$\overline{c}_k \equiv \frac{c_k - r_{0_k}}{d_k - r_{0_k}} \tag{4.4}$$

where $\overline{c}_k$ is the normalized response time of job J_k.

$$\Theta_{lb}(x) \equiv \max_{J_k \in JC} \overline{c}_k \tag{4.5}$$

Note that the above equations are the result of rewriting Equations (3.1) and (3.2) in terms of job parameters. A few remarks are due on computing the function $f_l(t)$ for job J_l in Algorithm A. As alluded in Step SA2 of Algorithm A, to minimize the system hazard, the function should express job's normalized response time $f_l(t) = \frac{t - r_{0_l}}{d_l - r_{0_l}}$. Note, however, that the system hazard (see Equation (4.5)) is affected only by the normalized response time of *completing* jobs. Thus, $f_l(t)$ needs to be calculated for these jobs only, and can be set to zero for the rest. Since Algorithm A is performed for one PN at a time, precedence constraints among tasks on different PNs are not accounted for, except for the way job arrival times are computed. This may result in a loose lower bound since the schedule is not "constrained" enough. A tighter lower bound can be obtained if such precedence constraints were accounted for in $f_l(t)$.

To derive $f_l(t)$ that considers precedence constraints among tasks on different PNs the notion of an *outgoing module* (OM) needs to be introduced. An OM, at search vertex x, refers to a module that has a precedence constraint (in the precedence constraint set PR) with another module in some *remote* task invocation. Typically, these are communication modules which send messages to a remote partner. A job representing such a module in the schedule is called an OM job. Completion of such jobs may enable modules on other PNs to execute. Consider an OM job, J_b, and let $C(J_b)$ denote the set of all completing jobs J_z preceded by J_b but assigned to a different PN at search vertex x. Let b_z be the length of the *critical path* from the OM job to the end of completing job J_z. It represents the minimum time elapsed from the completion of the OM job before the remote task invocation can complete.[2] The cost function $f_l(t)$ of J_l can be rewritten to account for such dependency as follows:

R1 **If** J_l is not a completing job and J_l is not an OM job **then** $f_l(t) = f_{l_1}(t) = 0$.

R2 **If** J_l is an OM job (but not a completing job) **then** $f_l(t) = f_{l_2}(t) = \max_{J_z \in C(J_l)} G_z(t)$, where $G_z(t) = (t + b_z - r_{0_z})/(d_z - r_{0_z})$.

R3 **If** J_l is completing job (but not an OM job) **then** $f_l(t) = f_{l_3}(t) = (t - r_{0_l})/(d_l - r_{0_l})$.

R4 **If** J_l is both a completing job and an OM job **then** $f_l(t) = f_{l_4}(t) = max\{f_{l_2}(t), f_{l_3}(t)\}$.

Once expressions for cost functions of modules residing on N_k are determined, Algorithm A can be applied to obtain the node hazard for each node. Vertex cost is set to the maximum computed node hazard. The computational complexity of deriving vertex cost is $O(nM^2)$, where n is the number of PNs and M the total number of modules in the system.

5 Examples

Two illustrative examples are presented in this section to demonstrate the power and utility of our task allocation algorithm. The examples are simplified for the purpose of clear demonstration of key ideas in the algorithm. In the first example, we consider the allocation of the workload shown in Figure 1. The second example is the allocation of part of a real turbofan engine control workload. In this example, the workload has been simplified and partitioned into tasks such that a task communicates only at the beginning and/or at the end of its execution. The examples mainly serve to illustrate how the algorithm works. The second example is to demonstrate the applicability of the proposed algorithm to a real-life problem. To obtain more experience with our algorithm's performance, a workload generator has been

[2] This is where network delays d_{ij} come into play.

M_j	e_{j1}	M_j	e_{j1}	M_j	e_{j1}
M_1	4	M_9	1:3	M_{17}	1:3
M_2	1:4	M_{10}	1:4	M_{18}	1
M_3	2	M_{11}	1	M_{19}	2
M_4	2	M_{12}	2:4	M_{20}	0:1
M_5	2:6	M_{13}	2	M_{21}	1:3
M_6	2	M_{14}	0:2	M_{22}	1
M_7	1	M_{15}	2:3	M_{23}	2
M_8	1:2	M_{16}	3	M_{24}	1:2

Table 2: Module execution times on N_1 in Example 1.

constructed to create task sets of arbitrary size. The computational results obtained from running the algorithm on the created task sets are presented is Section 6.

5.1 Example 1

Consider an example of allocating the three tasks T_1, T_2, and T_3 in Figure 1 to two processors N_1 and N_2. Within the planning cycle $[0, 40)$, T_1 and T_2, both with period 40, are invoked only once while T_3, with period 20, is invoked twice. Assume the deadline of each task invocation is at the end of its period. The execution times of various modules are shown in Table 2 for processor N_1. The execution times of communication modules are given in the format $e_{jq_{local}} : e_{jq_{remote}}$. For simplicity we assume that the execution times on N_2 are exactly a half of those on N_1. In general, the ratio between module execution times on two processors may be different for different modules since it depends on the instruction mix. For example, if one of the processors has a more powerful floating-point unit then floating-point instructions will execute faster while, say, memory access instructions may take the same amount of time on both.

Figure 4 shows all generated search vertices numbered in the order of their generation times. The assignment and cost, $\Theta(x)$, associated with each vertex x are also indicated in Figure 4. In this example, two leaf vertices are generated before our B&B algorithm finds an optimal solution. Specifically, vertex 6, with $\Theta(6) = 39/40$, is eliminated first as soon as vertex 8 with the exact cost $28/40 < \Theta(6)$ is generated. Then, all active vertices 4, 5, and 8 are eliminated after vertex 9 is generated since the system hazard of this complete assignment, 23.5/40, is the smallest. Thus, vertex 9, which assigns all three tasks to N_2, is an optimal solution to the allocation problem, and its optimal schedule is shown in Figure 5.

The optimal allocation assigns all three tasks to the same PN. One reason for this is that communication modules and delays in this example are the major part of the task load, and thus, assigning all tasks to a single PN is the best. Note also that the tasks have been assigned to the faster PN. To see

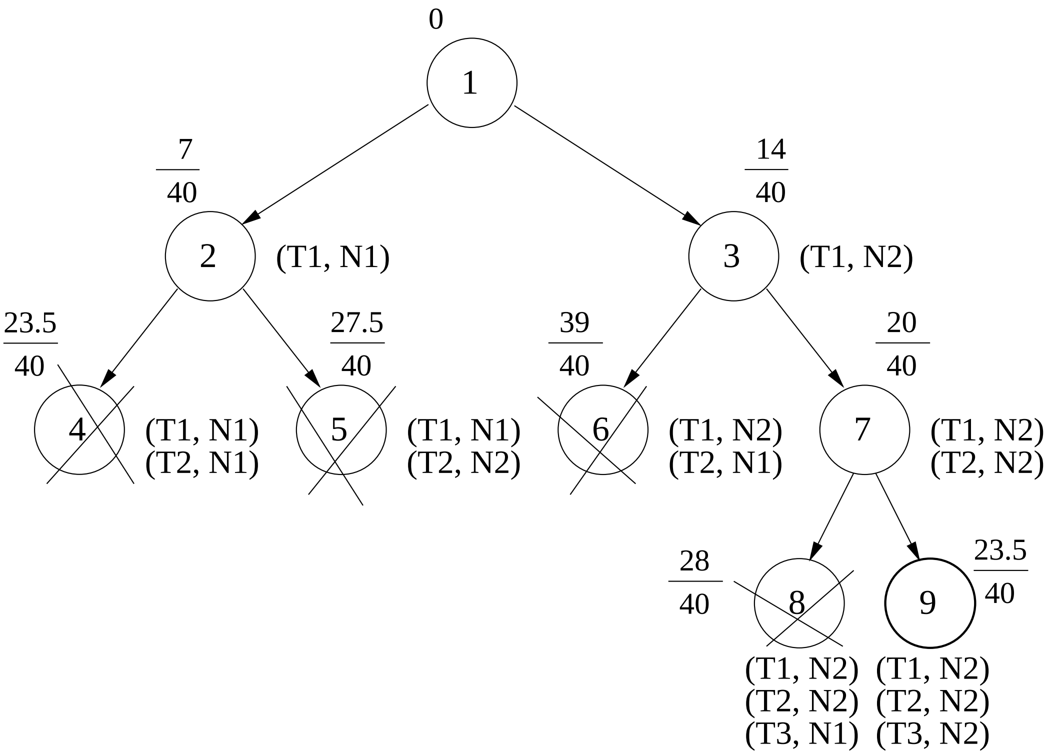

Figure 4: The search tree generated for example 1.

how $\Theta(x)$ is obtained for a non-terminal vertex x, consider the computation of $\Theta(5)$. Figure 4 shows that at vertex 5 task T_1 is assigned to N_1, T_2 is assigned to N_2 while task T_3 has not been assigned. Figure 6 redraws the task graph in Figure 1, showing the execution times of allocated modules and communication delays computed for this particular allocation (from data in Table 2) as described in Section 4.2.1. Invocations of the unallocated task T_3 are represented, as described in Section 4.2.2, by a lumped execution time ρ_q for each N_q. For a particular task invocation, ρ_q denotes the minimim load the invocation imposes on N_q at any solution descending from vertex 5.

To compute $\Theta(5)$, first the load of allocated tasks (i.e., modules $M_1, ..., M_{16}$) is computed. M_2, M_5, M_{10} and M_{12} are considered to communicate remotely (i.e., their execution time is $e_{j_{remote}}$), since they and their communication partners are assigned to different PNs. Modules M_8, M_9, M_{14} and M_{15} are, by default, considered to communicate locally since their communication partners are not yet allocated. The rest are computation modules whose execution times, e_{jq} are fixed by their assignment. Figure 6 gives the resulting module execution times. For purposes of scheduling, a job is associated with each allocated module M_j. A job representing M_j has the execution time shown for M_j in Figure 6 and the deadline of the task invocation M_j. For example, the deadline of (the job representing) M_1 is that of task invocation T_{11}, which is 40. Jobs' release times are computed from Equation (4.1). For example, the release time of (the job representing) M_3 is zero, and the release time of (the job representing) M_{16}

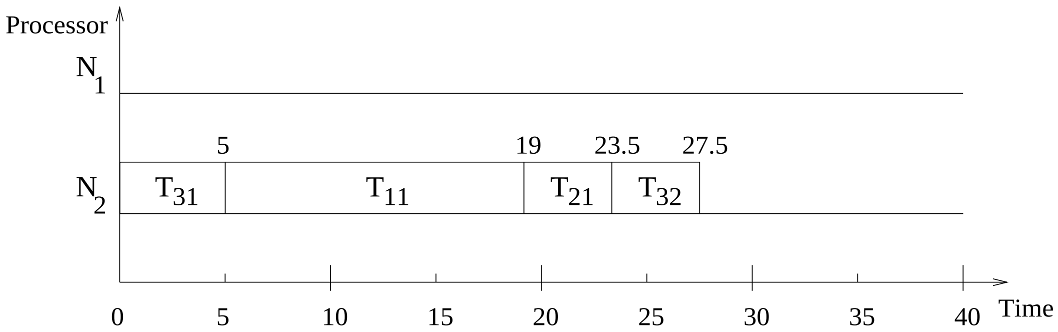

Figure 5: An optimal schedule for example 1.

is 20, because the latter is preceded by modules in task invocation T_{32}, which is released at $r_{32} = 20$.

Next, the minimum load imposed on each PN by unassigned tasks, in this case only task T_3, is computed. T_3 has two invocations, T_{31} and T_{32}, within a planning cycle. Thus, we need to compute loads $\rho_{1,T_{31}}$ and $\rho_{1,T_{32}}$ (imposed on N_1) and loads $\rho_{2,T_{31}}$ and $\rho_{2,T_{32}}$ (imposed on N_2) using Equation (4.2). Since neither invocation communicates with modules on N_1, the minimum load imposed on N_1 due to T_3 is zero (which corresponds to assigning T_3 to N_2). In other words, $\rho_{1,T_{31}}$ and $\rho_{1,T_{32}}$ are zero because the summation in the second argument of the *min* function in Equation (4.2) is vacuous. To compute the minimum load imposed on N_2, note that the sets $\psi_{2,T_{31}}$ and $\psi_{2,T_{32}}$ of communication modules on N_2 preceding the completion of T_{31} and T_{32} are $\{M_8\}$, and $\{M_{14}\}$, respectively. Substituting in Equation (4.2), the load imposed on N_2 by T_{31} is $\rho_{2,T_{31}} = \min\{4, 0.5\} = 0.5$, where 4 is the sum of execution times of T_{31}'s modules on N_2, and 0.5 is the increase in the execution time of M_8, which precedes T_{31}'s completion. Similarly, $\rho_{2,T_{32}} = \min\{5, 1\} = 1$, where 5 is the sum of execution times of T_{32}'s modules on N_2, and 1 is the increase in the execution time of M_{14}, which precedes T_{32}'s completion. For the purpose of scheduling, a job is associated with each unallocated invocation T_{iv}, in this case T_{31} and T_{32}. As described in Section 4.2.2 the jobs' execution times are set to $\rho_{2,T_{31}}$ and $\rho_{2,T_{32}}$, respectively, and their deadlines are set to the corresponding invocation deadline. The release time of both jobs are zero, as computed from Equation (4.3). This is self-evident for T_{31} and is true of T_{32} because some of its modules are preceded by the modules of T_2 which is released at time zero.

Having computed the job set for allocated and unallocated tasks, the third and final step is to compute the cost functions $f_l(t)$ of each job J_l and schedule these jobs using Algorithm A. To find the cost functions, rules R1-R4 of Section 4.2.3 are followed. For example, consider the jobs representing the modules of the allocated task T_1. (For simplicity we will use module names to denote the corresponding jobs.) From R3 the cost function of M_4 and M_5 is $t/40$. From R1, the cost functions of M_1 and M_3 are 0. Finally, the cost function of M_2 is determined by considering the critical paths from M_2 to the completion of task invocations T_{21} and T_{32} which depend on M_2. Let these paths be of lengths b_1 and b_2, respectively. Thus, from R2, the cost function of M_2 is $\max\{(t+b_1-0)/40, (t+b_2-0)/40\} = (t+13)/40$.

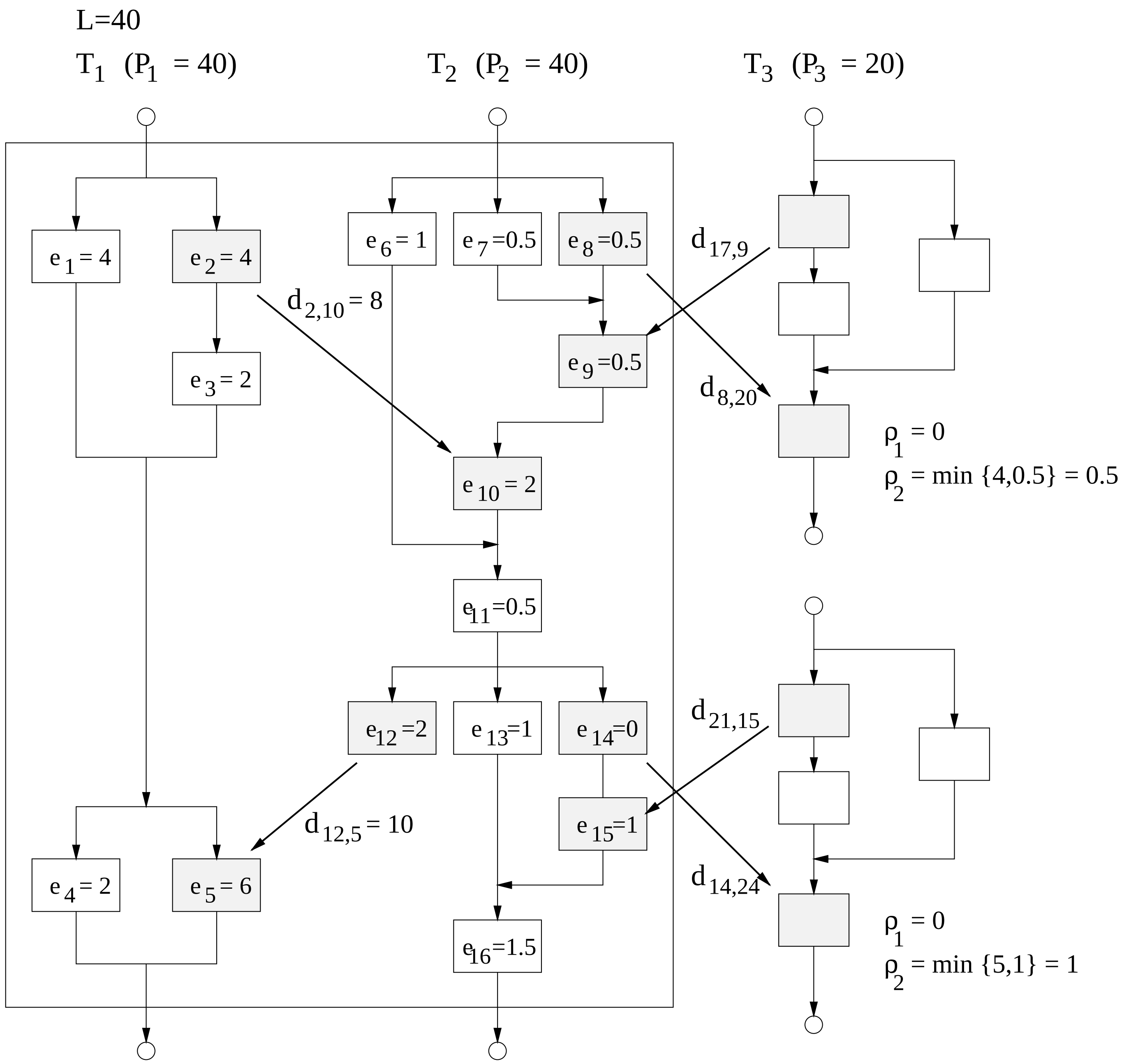

Figure 6: Task graph at vertex 5.

Cost functions of jobs representing unallocated task invocations are computed similarly. In this case, R3 yields the cost function $(t-0)/20$ for T_{31} and the cost function $(t-20)/20$ for T_{32}. As a result of applying Algorithm A, the node hazard of the obtained schedule is 27.5/40 for N_1 and 20/40 for N_2. Therefore, $\Theta(5) = 27.5/40$.

5.2 Example 2

We consider allocating parts of a real turbofan engine control program. The workload has been simplified and partitioned into 13 tasks, $T_1, \cdots, T_{13}$ all of which have an identical period of 300 time units and communicate with one another only at the beginning and/or at the end of, but not during, their execution. Each task is partitioned into three modules α, β, and γ, where α represents the sum of all execution times associated with pure computation of the task, while β and γ denote the the sums of all execution times associated with processing incoming and outgoing messages, respectively. For simplicity, the following assumptions are made.

- We assume that the amount of communication between any task T_j and each of its immediate predecessors is the same. Similary, we assume that the amount of communication between the task and each of its successors is equal. Therefore, if a fraction ϕ_1 of T_j's predecessors and a fraction ϕ_2 of T_j's successors are assigned to PNs where T_j does not reside, then the total execution time of T_j is computed as $\alpha + \phi_1\beta + \phi_2\gamma$. (More precisely, $E_{jq} = \alpha_{jq} + \phi_1\beta_{jq} + \phi_2\gamma_{jq}$, where α_{jq}, β_{jq}, and γ_{jq} are the aforementioned components of execution time measured for T_j on processor N_q.
- The communication delay d_{ij} between any two tasks assigned to different PNs is constant for all task pairs and equal to 2 time units.
- A task is considered *completed* only after all of its computation and communication modules have been completed.

Consider the allocation of the above workload to a distributed system of two PNs where N_2 is twice faster than N_1. The task graph and task execution times (on N_1) are shown in Figure 7. Since the invocation periods of all tasks are the same, the optimal allocation w.r.t. the system hazard is the same as that w.r.t. the makespan or the maximum task completion time among all tasks. The optimal allocation is found after visiting only 81, out of a total of $2^{14} - 1 = 16383$ vertices in the B&B search tree. The optimal allocation assigns T_1 and $T_3, ..., T_9$ to the faster processor and the rest to the slower one. The minimum system hazard achieved is $183/300 = 0.61$. Note that the reduction of search tree observed above may be attributed to the simplicity of the example. In the following section we explore algorithm efficiency for more complex systems.

6 Computational Experiences

In order to test the performance of the algorithm with larger task systems a workload generator has been constructed to create task sets of arbitrary size. The generator creates a specified number of tasks, with a given average number of invocations per task, and a given average number of modules per invocation (Poisson-distributed), then constructs the task graph by generating precedence constraints (with no directed cycles) among modules. Precedence constraints created between pairs of modules in the same

Task	Name	α/β/γ
T1	FULL DUCT	40/0/4
T2	LPFP	26/2/0
T3	HPFP	168/2/24
T4	MFV. DIFF.	34/24/12
T5	DWCOMERS	24/4/3
T6	LO. NOZZ.	23/3/2
T7	NOZZ. FLOW	24/2/4
T8	MIXER	23/4/1
T9	SP. DUCT	24/1/0
T10	MCC. DUCT	27/4/4
T11	B. THROAT	23/4/1
T12	A. THROAT	23/1/0
T13	CCV. DUCT	23/4/0

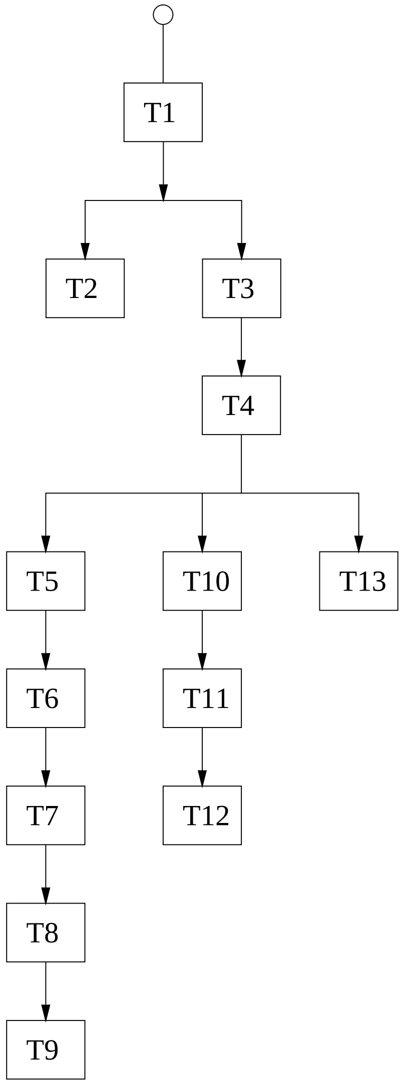

Figure 7: Task graph for Example 2

task represent the sequence of computation in that task. Precedence constraints created between pairs of modules of different tasks result from intertask communications. Module types are selected accordingly, and delays are inserted in their appropriate places between communicating modules. Module execution times are represented by a Poisson-distributed random variable of a specified average. The number of precedence constraints is approximately the number of modules in the task graph. The number of communicating task pairs is approximately 1 to 1.5 times the number of created tasks.

To assess algorithm performance with variable task system sizes, 50 task sets were generated and divided into five categories of 6-task, 8-task, 10-task, 12-task and 14-task sets with an average of 60, 80, 100, 120 and 140 modules per set, respectively. A homogeneous 4-PN architecture, and fixed message

Tasks	Expanded Vertices	Total Space	% Expanded
6	18	4096	0.43
8	65	65536	0.1
10	95	1048576	0.01
12	133	16777216	0.0008
14	274	268435456	0.0000004

Table 3: Algorithm performance for different task-set sizes.

size was assumed. The B&B algorithm was applied to each task set, and the number of expanded vertices was recorded in each run. Table 3 summarizes the obtained results. It can be seen that the heuristic employed at non-terminal search vertices guides the algorithm efficiently for an optimal solution. For example, it takes less than 100 vertices to find an optimal allocation of 10 tasks (100 modules) on 4 PNs, which is less than 0.01 % of the total search space, and takes less than 300 vertices to find an optimal allocation of 14 tasks, which is about 4×10^{-7} of the total search space.

To explore the performance of the algorithm for different number of PNs, 10 task sets were generated, each with 8 tasks and 80 modules. The algorithm was used to allocate the tasks on homogeneous 2-PN, 4-PN and 6-PN architectures, and the number of expanded vertices was recorded in each run. Table 4 summarizes the obtained results. It can be seen, from Table 4, that less than 40 vertices are expanded on average in all cases. It can also be seen that the average number of expanded vertices is almost the same for 4- and 6- PN architectures. The relative "saturation" in the number of expanded vertices with respect to the number of PNs in the system is because when the number of available PNs exceeds that required by the optimal allocation, the heuristic tends to guide the search away from expanding vertices which exploit a larger (than necessary) number of PNs. Therefore, the subset of searched vertices before a solution is found tends to remain the same.

Note that the purpose of the above experiments was not to accurately quantify the behavior of the algorithm, but rather to verify its qualitative performance trends. Thus, for example, we conducted only 10 experiments per data point in Tables 3 and 4. The conclusion we draw from the above experimental observations is that the algorithm has potential to find optimal solutions efficiently. In particular, the diminishing percentage of expanded vertices relative to total search space shows that the pruning function is efficient. More accurate analysis of algorithm performance is not meaningful without a more representative load generation technique that reflects more faithfully the structure of real applications of interest. Such analysis and techniques, however, are outside the scope of this paper. In general, the algorithm is expected to work best when intertask communication is considerable and when communication costs are comparable to computation costs. A careful analysis of the way the load of unallocated task is computed in Section 4.2.2 (see Equation (4.2)) can show that if communication is insignificant $\psi_{q,T_{iv}}$ tends to be vacuous, which causes this load to be inadequately represented. As a result, the computed lower bounds are loose and a larger number of vertices are expanded. However, this is not a

Nodes	Expanded Vertices	Coef of Variation	Total Space	% Expanded
2	16	0.35	256	6.17
4	37	0.8	65536	0.06
6	38	0.8	1679616	0.002

Table 4: Algorithm performance for different number of PNs.

disadvantage of our algorithm. When tasks tend not to communicate, many of earlier research results on task allocation are applicable. The difficult challenge addressed in this paper is the case when intertask communication plays an important role. To give an example of performance improvement with increasing intertask communication, let's compare Table 3 with Table 4. The latter table is computed for a set of experiments where the number of communicating task pairs was approximately 1.5 times that in the former, thus representing more heavily communicating tasks. Table 3 shows that for 8 tasks on a 4-processor system the average number of expanded vertices was 65. Table 4, on the other hand, shows that for 8 tasks on a 4-processor system the average number of expanded vertices was only 37. This comparison[3] supports our theoretical expectations regarding algorithm performance.

7 Conclusions

Task allocation is one of the most important issues in designing distributed real-time systems. However, this problem is generally known to be NP-hard even without considering the precedence constraints resulting from inter-task communications. In this paper, we have addressed the problem of allocating a set of communicating periodic tasks to the processing nodes of a distributed real-time system. We solved this problem by using a B&B algorithm which employs a new bounding function (at non-leaf vertices) to guide the search to the optimal allocation. The proposed solution can be easily extended to the case where tasks are not periodic. Note that the periodicity assumption has been used *only* to derive the length of the planning cycle and the release times of task invocations. The rest of the paper (as well as the scheduling algorithms [1, 55]) does not rely on this assumption. Furthermore, our model can be easily extended to the case where modules, rather than tasks, are the objects to be allocated.

The lower-bound costs presented in this paper are shown to reduce the computational difficulty significantly. Therefore, the proposed task allocation algorithm has high potential for practical use. This fact has been confirmed by our computational experiences.

Because of the enumerative nature of the proposed B&B algorithm, it is also possible to apply it to task allocation problems with other resource constraints, such as those on memory and communication bandwidths.

[3]The authors would like to thank the anonymous reviewer of an earlier manuscript of this paper for pointing out this observation from data in the two tables.

References

[1] D. Peng and K. G. Shin, "Optimal scheduling of cooperative tasks in a distributed system using an enumerative method," *IEEE Trans Soft. Eng.*, vol. 19, no. 3, pp. 253–267, March 1993.

[2] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, January 1995.

[3] K. G. Shin and Y.-C. Chang, "Load sharing in distributed real-time systems with state-change broadcasts," *IEEE Trans. on Comput.*, vol. C-38, no. 8, pp. 1124–1142, August 1989.

[4] K. Goswami, M. Devarakonda, and R. Lyer, "Prediction-based dynamic load-sharing heuristics," *IEEE Trans. on Parallel and Distrib. Sys.*, vol. 4, no. 6, pp. 638–648, June 1993.

[5] K. G. Shin and C.-J. Hou, "Analytic models of adaptive load sharing schemes in distributed real-time systems," *IEEE Trans. Paralled and Distributed Systems*, vol. 4, no. 7, pp. 740–761, July 1993.

[6] K. R. Baker, *Introduction to Sequencing and Scheduling*, Wiley & Sons, 1974.

[7] W. W. Chu, "Task allocation in distributed data processing," *IEEE Computer*, vol. 13, pp. 57–69, Nov 1980.

[8] W. W. Chu and L. M. Lan, "Task allocation and precedence relations for distributed real-time systems," *IEEE Trans. on Computers*, vol. C-36, no. 6, pp. 667–679, June 1987.

[9] S. French, *Sequencing and Scheduling*, Halsted Press, 1982.

[10] R. E. D. Woolsey and H. S. Swanson, *Operations Research for Immediate Applications: A Quick and Dirty Manual*, Harper and Row, 1974.

[11] S. Selvakumar and C. S. R. Murthy, "Static task allocation of concurrent programs for distributed computing systems with processor and resource heterogeneity," *Parallel Computing*, vol. 20, no. 6, pp. 835–851, 1994.

[12] H. H. Ali and H. El-Rewini, "Task allocation in distributed systems: a split graph model," *J. Combin. Math. Combin. Comput.*, vol. 14, no. 1, pp. 15–32, January 1993.

[13] A. Billionnet, M.-C. Costa, and A. Sutter, "An efficient algorithm for a task allocation problem," *J. Assoc. Comput. Mach.*, vol. 39, no. 3, pp. 502–518, March 1992.

[14] S. H. Bokhari, "A network flow model for load balancing in circuit-switched multicomputers," *IEEE Trans. Parallel and Distrib. Sys.*, vol. 4, no. 6, pp. 649–657, June 1993.

[15] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[16] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. of ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[17] O. Serlin, "Scheduling of time critical processes," in *Proc. of AFIPS 1972 Spring Joint Computer Conf.*, pp. 925–932, Montvale, N. J., 1972, AFIPS Press.

[18] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithm," *IEEE Trans. on Software Engineering*, vol. SE-3, no. 1, pp. 85–93, January 1977.

[19] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *IEEE Computer*, vol. 11, pp. 97–106, July 1978.

[20] P. Y. R. Ma and et. al., "A task allocation model for distributed computing systems," *IEEE Trans. on Computers*, vol. C-31, no. 1, pp. 41–47, January 1982.

[21] C. C. Shen and W. H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Trans. on Computers*, vol. C-34, no. 3, pp. 197–203, March 1985.

[22] J. B. Sinclair, "Efficient computation of optimal assignments for distributed tasks," *J. of Parallel and Distributed Computing*, vol. 4, pp. 342–362, 1987.

[23] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Trans. Comput.*, vol. 37, no. 11, pp. 1384–1397, November 1988.

[24] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley and Sons, New York, 1976.

[25] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[26] J. K. Lenstra and A. H. G. R. Kan, "Complexity of scheduling under precedence constraints," *Operations Research*, vol. 26, no. 1, pp. 23–35, Jan 1978.

[27] E. L. Lawler, *Deterministic and Stochastic Scheduling*, chapter Recent Developments in Deterministic Sequencing and Scheduling: A Survey, pp. 35–74, Reidel, Dordrecht, The Netherlands, 1982.

[28] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. on Computers*, vol. C-33, no. 11, pp. 1023–1029, November 1984.

[29] W. W. Chu and K. Leung, "Module replication and assignment for real-time distributed processing systems," *Proc of IEEE*, vol. 75, no. 5, pp. 547–562, May 1987.

[30] W. H. Kohler and K. Steiglitz, *Computer and Job-Shop Scheduling Theory*, chapter Enumerative and Iterative Computational Approach, pp. 229–287, Wiley and Sons, 1976.

[31] M. Alfano, A. Di-Stefano, L. Lo-Bello, O. Mirabella, and J. H. Stewman, "An expert system for planning real-time distributed task allocation," in *Proceedings of the Florida AI Research Symposium*, Key West, FL, USA, May 1996.

[32] P. Altenbernd, C. Ditze, P. Laplante, and W. Halang, "Allocation of periodic real-time tasks," in *20th IFAC/IFIP Workshop*, Fort Lauderdale, FL, USA, November 1995.

[33] J. L. Lanet, "Task allocation in a hard real-time distributed system," in *Proceedings of the 2nd Conference on Real-Time Systems*, pp. 244–252, Szlarska Poreba, Poland, September 1995.

[34] T. C. Lueth and T. Laengle, "Task description, decomposition and allocation in a distributed autonomous multi-agent robot system," in *Proceedings of International Conference on Intelligent Robots and Systems*, pp. 1516–1523, Munich, Germany, September 1994.

[35] C. M. Hopper and Y. Pan, "Task allocation in distributed computer systems through an ai planner solver," in *Proceedings of IEEE 1995 National Aerospace and Electronics Conference*, volume 2, pp. 610–616, Dayton, OH, USA, May 1995.

[36] B. R. Tsai and K. G. Shin, "Assignment of task modules in hypercube multicomputers with component failures for communication efficiency," *IEEE Transactions on Computers*, vol. C-43, no. 5, pp. 613–618, May 1994.

[37] K. G. Shin and C. J. Hou, "Evaluation of load sharing in harts with consideration of its communication activities," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 7, pp. 724–739, July 1996.

[38] S. M. Yoo and H. Y. Youn, "An efficient task allocation scheme for two dimensional mesh-connected systems," in *Proceedings of the 15th International Conference on Distributed Computing Systems*, pp. 501–508, Vancouver, Canada, 1995.

[39] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[40] K. Tindell, A. Burns, and A. Wellings, "Allocating hard real-time tasks: An np-hard problem made easy," *J. of Real-Time Systems*, vol. 4, no. 2, pp. 145–166, May 1992.

[41] E. Wells and C. C. Caroll, "An augmented approach to task allocation: Combining simulated annealing with list-based heuristics," in *Proc. Euromicro Workshop*, pp. 508–515, 1993.

[42] J. E. Beck and D. P. Siewiorek, "Simulated annealing applied to multicomputer task allocation and processor specification," in *Proceedings of 8th IEEE Symposium on Parallel and Distributed Processing*, pp. 232–239, October 1996.

[43] S. T. Cheng, S. I. Hwang, and A. K. Agrawala, "Schedulability oriented replication of periodic tasks in distributed real-time systems," in *Proceedings of the 15th International Conference on Distributed Computing Systems*, Vancouver, Canada, 1995.

[44] S. B. Shukla and D. P. Agrawal, "A framework for mapping periodic real-time applications on multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 7, pp. 778–784, July 1994.

[45] T.-S. Tia and J. W.-S. Liu, "Assigning real-time tasks and resources to distributed systems," *Internnational Journal of Minim and Microcomputers*, vol. 17, no. 1, pp. 18–25, 1995.

[46] S. S. Wu and D. Sweeping, "Heuristic algorithms for task assignment and scheduling in a processor network," *Parallel Computing*, vol. 20, pp. 1–14, 1994.

[47] K. Ramamritham, "Allocation and scheduling of precedence-related periodic tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412–420, April 1995.

[48] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *IEEE Transactions on Software Engineering*, vol. 19, no. 2, pp. 139–154, February 1993.

[49] P. Scholz and E. Harbeck, "Task assignment for distributed computing," in *Proceedings of the 1997 Conference on Advances in Parallel and Distributed Computing*, pp. 270–277, Shanghai, China, March 1997.

[50] Y. Oh and S. H. Son, "Scheduling hard real-time tasks with tolerance to multiple processor failures," *Multiprocessing and Multiprogramming*, vol. 40, pp. 193–206, 1994.

[51] C.-J. Hou and K. G. Shin, "Replication and allocation of task modules in distributed real-time systems," in *24th IEEE Symposuim on Fault-Tolerant Computing Systems*, pp. 26–35, June 1994.

[52] K. G. Shin and S. Daniel, "Analysis and implementation of hybrid switching," *IEEE Trans. on Computers*, pp. 211–219, 1995.

[53] D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multihop networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[54] D.-T. Peng and K. G. Shin, "A new performance measure for scheduling independent real-time tasks," *J. Parallel Distrib. Comput.*, vol. 19, no. 1, pp. 11–26, 1993.

[55] K. R. Baker et al., "Preemptive scheduling of a single machine to minimize maximum cost subject to release dates and precedence constraints," *Operations Research*, vol. 31, no. 2, pp. 381–386, March 1983.