4

RADC-TR-88-147
Final Technical Report
March 1989

BM/C^3 ALGORITHM MAPPING ONTO CONCURRENT PROCESSORS

University of Connecticut

Sponsored by
Strategic Defense Initiative Office

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

BM/C^3 ALGORITHM MAPPING ONTO CONCURRENT PROCESSORS

Krishna R. Pattipati
Peter B. Luh
Rong-Tay Lee
Samir Shah
Somnath Deb

Contractor: University of Connecticut
Contract Number: F30602-81-C-0169
Effective Date of Contract: 4 May 87
Contract Expiration Date: 30 Dec 87
Short Title of Work: BMC^3 Algorithm Mapping Onto Concurrent Processors

Period of Work Covered: May 87 - Dec 87

Principal Investigator: Krishna Pattipati
Phone: (203) 486-2890

Project Engineer: Patrick J. O'Neill
Phone: (315) 330-2925

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by Patrick J. O'Neill RADC (COTC), Griffiss AFB NY 13441-5700, under Contract F30602-81-C-0169.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-88-3	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-147
6a. NAME OF PERFORMING ORGANIZATION University of Connecticut; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center
6c. ADDRESS (City, State, and ZIP Code) Department of Electrical & Systems Engineering Storrs CT 06268	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Strategic Defense Initiative Office; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0169
8c. ADDRESS (City, State, and ZIP Code) Office of the Secretary of Defense Wash DC 20301-7100	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 63223C; PROJECT NO. B413; TASK NO 03; WORK UNIT ACCESSION NO P5

11. TITLE (Include Security Classification)
BM/C^3 ALGORITHM MAPPING ONTO CONCURRENT PROCESSORS

12. PERSONAL AUTHOR(S)
Krishna R. Pattipati, Peter B. Luh, Rong-Tay Lee, Samir Shah, Somnath Deb

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM May 87 TO Dec 87	March 1989	102

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES FIELD	GROUP	SUB-GROUP	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
12	07		SDI; BMC^3 Algorithms; Mapping Algorithms; Parallel Processing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report is concerned with the mapping of large scale resource allocation algorithms onto parallel computing architectures. The mapping problem is viewed as one of assigning the nodes of a finite, directed, acyclic task graph (representing the logical and data dependencies among the tasks constituting the algorithm) onto the nodes of a finite, undirected processor graph (denoting the parallel computing architecture). The objective is to minimize the completion time of the algorithm such that the redundancy, processor memory and security constraints are satisfied. The delays introduced by task queueing, message transmission, message collision and precedence constraints are explicitly modeled. We present four algorithms to solve the mapping problem. The first algorithm is a two-stage heuristic that determines the order of task execution based on the critical path method (CPM) and then employs a one-step optimization method to determine the task allocation. The second algorithm employs the idea of pair-wise exchange on task execution order to

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Patrick J. O'Neill	22b. TELEPHONE (Include Area Code) (315) 330-2925; 22c. OFFICE SYMBOL RADC (COTC)

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Block 19. Abstract (Cont'd)

improve the heuristic algorithm. The third algorithm is an optimal mapping strategy using A* algorithm, wherein a lower bound on the cost-to-go is obtained from the two-stage heuristic. Finally, an approximation algorithm based on the idea of A* is developed. Extensive computational experiments on hundreds of random graphs have shown that the heuristic algorithm provides optimal mapping whenever the ratio of computation/communication is very high or very small, and that the pair-wise exchange algorithm provides uniformly good mapping for all values of the ratios. Hypothetical examples and applications to weapon target assignment and multi-target tracking problems are also included to show the effectiveness of the mapping algorithms. In addition, the mapping algorithms have been integrated into an interactive software package, termed MAPPER, for the analysis and performance evaluation of alternative BM/C3 algorithms.

UNCLASSIFIED

Table of Contents

List of Figures

List of Tables

SECTION 1
INTRODUCTION

1.1 PROBLEM SIGNIFICANCE

The recent advances in very large scale integration technology and communication networks have generated a great deal of interest in the study of multiple-instruction, multiple-data stream (MIMD) multi-processor systems. These systems are characterized by: (1) a number of autonomous processing elements interconnected by a high-bandwidth communication network, (2) a distributed operating system, and (3) highly concurrent computation brought about by the decomposition of an application algorithm into several distinct, cooperating tasks [Kuhl and Reddy, 1986]. In addition, the multiplicity of processing elements in a multi-processor system can be exploited to improve system reliability, and to provide graceful degradation in the presence of hardware and software faults. The modularity, flexibility, and reliability of these systems make them attractive to many areas of real-time applications, such as large scale defense applications, flight control systems, transportation systems, and manufacturing. These applications typically have enormous computational and storage requirements, real-time processing constraints, fault-tolerance and data security requirements [Pattipati et al., 1986].

One of the major issues in the efficient operation of a MIMD multi-processor is the mapping of an application algorithm onto various constituent processors of the system. In order to take advantage of concurrent processing, it is desired to achieve a minimum execution time (completion time) of the algorithm with a minimum number of processing elements via efficient algorithm-architecture mapping. There are four important factors that contribute to the completion time of an algorithm on a MIMD multi-processor: (1) task partitioning, (2) task allocation and sequencing, (3) the interconnection topology and the capacity of each communication link, and (4) the speed of each processor. The partitioning problem refers to the selection of the level of granularity used to represent the task of an application algorithm. Task partitioning determines the amount of computation required by each task, the precedence relations among the tasks of the algorithm, and the amount of data transmitted between each pair of tasks. The allocation and sequencing strategy determines the assignment of the set of tasks to member processors and the order of execution of the tasks allocated to each processor. The inter-task communication is constrainted by the interconnection topology, and the precedence relations among the tasks of the algorithm impose synchronization requirements, i.e., a task can not begin executing until all the tasks preceding it have been completed.

In this report, we are concerned with the problem of mapping Battle Management/Command, Control, and Communication (BM/C^3) algorithms for multi-target tracking and weapon-target assignment onto a non-homogeneous MIMD multiprocessor to minimize the completion time (or equivalently, maximize speedup). We view the mapping problem as one of characterizing a BM/C^3 algorithm and the multiprocessor system as graphs, and subsequently assigning and sequencing the nodes of the algorithm graph to the nodes of the processor graph to minimize the completion time of the algorithm, subject to reliability, storage, and security constraints. A flow chart of the mapping process is shown in Fig. 1-1. A BM/C^3 algorithm is partitioned into tasks, and the communication among tasks is represented as a directed acyclic graph. These graphs are termed task graphs, problem graphs, or computation flow graphs. A multi-processor, on the other hand, is characterized by an undirected graph that depicts the interconnection topology of the architecture. These graphs are termed the processor graphs, system graphs, or computation resource graphs. In the BM/C^3 application, the task and processor graphs are time-varying due to: (1) the dynamic nature of communication among tasks, (2) lack of a priori information on the data dependencies among tasks, (3) the stochastic nature of the time between the execution of a given task, and (4) failures and/or on-line repair of the computational resources of the multi-processor architecture. As discussed below, various assumptions on the time-dependence of the task and processor graph parameters lead to static, quasi-static, and dynamic mapping problems. The resulting optimization problem of allocating tasks to processors, and sequencing the tasks on member processors is solved, and the performance measures such as the processor utilization, completion time, speedup, and communication delay are computed. If the results are not satisfactory, an alternative task division or a new multi-processor architecture may be tried out, and the analysis repeated.

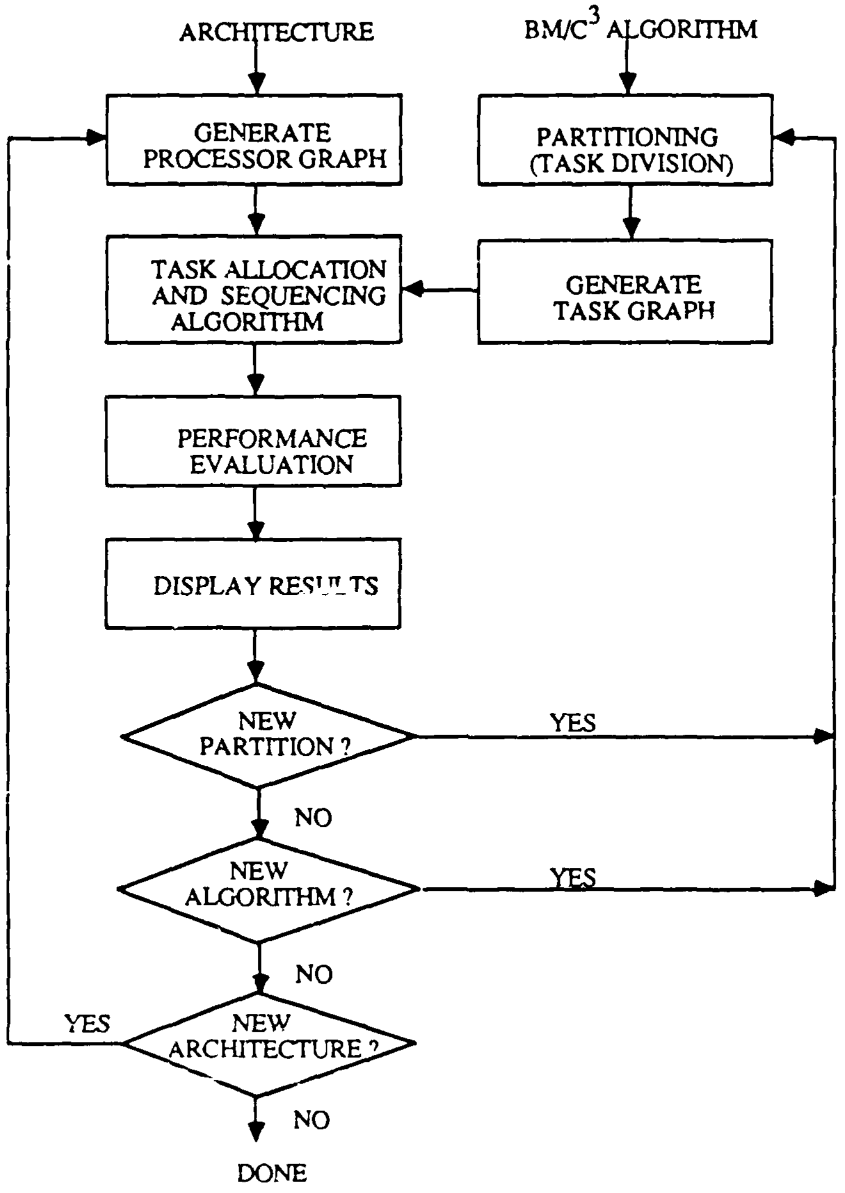


FIGURE 1-1: OVERVIEW OF ALGORITHM-ARCHITECTURE MAPPING PROBLEM

1.2 A TAXONOMY OF MAPPING PROBLEMS

The mapping problems can be classified based on the following two elements: (1) the *time-dependence* of the task and processor graph parameters, and (2) the *level of information* on the task graph parameters. For the multi-processor configuration, the amount of task computation and communication among tasks can be stationary (static or time-invariant) or dynamic. Dynamic behavior refers to a situation wherein the task graph and/or processor graph parameters are time varying, while in the static case they remain fixed. This in turn gives rise to static and dynamic mapping methods. The dynamic mapping methods are considerably more complex than the static mapping problems, since they involve multi-stage optimization, wherein the effects of current mapping decisions on all future changes in the task and processor graphs must be considered. In some cases, the computation and communication vary slowly with time, and, hence, can be assumed to be static over relatively long time intervals. We call such task graphs quasi-static, and the corresponding mapping methods are termed quasi-static mapping methods. In this case, we solve a series of static mapping problems periodically by taking into account the migration cost incurred in changing the current mapping to the next mapping. The quasi-static mapping methods are also useful when the task graphs are relatively stationary, but the multi-processor system might undergo configuration changes due to failures and recovery. In this case, we solve a series of static mapping problems for each possible multi-processor configuration or a small set of aggregated configurations. As mentioned earlier, the data dependencies in BM/C^3 algorithms fall under the dynamic category, but can be approximated by a quasi-static model.

The task parameters can be either deterministic or probabilistic. In the deterministic situation, the amount of task computation, and the amount and frequency of data transfers among tasks are perfectly known. On the other hand, the probabilistic knowledge of the task graph parameters can assume one of two forms : "complete information" or "partial information". Under complete information, the probabilistic knowledge of the task parameters is represented by a known probability distribution function with known parameters of the distribution. For example, if the amount of data transferred between two tasks is exponentially distributed with a given mean, then the information is considered complete. Under partial information, the form of the probability densities of the task parameters is known, but the parameters of the distribution have to be estimated (learned) on-line. For example, we may know that the amount of data transferred between two tasks is exponentially distributed, but the mean is unknown.

The classification discussed above is summarized in Fig. 1-2. Our primary focus in this report is on the static, deterministic mapping problem that explicitly considers the precedence restrictions on task execution, data communication delay, and redundancy, security, and storage constraints. Future research will address the quasi-static and dynamic mapping problems.

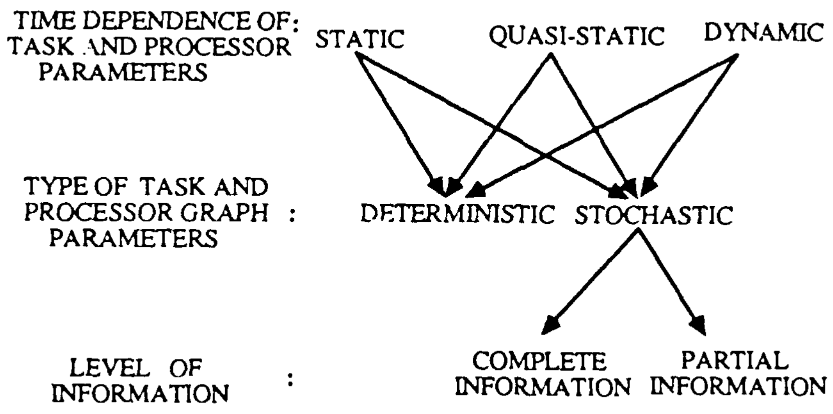


FIGURE 1-2 : A TAXONOMY OF MAPPING PROBLEMS

1.3 SUMMARY OF RESULTS

The static, deterministic mapping problem is NP-complete, i.e., the memory and computational requirements of an optimal algorithm grow exponentially with the number of tasks and the number of processors [Horowitz and Sahni, 1978; Garey and Johnson, 1979; Bokhari, 1981]. Therefore, all practical algorithms involve the use of heuristics to subdue the computational explosion of the optimal mapping algorithms. We develop four algorithms to solve the mapping problem: (1) greedy heuristic algorithm, (2) pair-wise exchange algorithm, (3) optimal A^* algorithm, and (4) ε-optimal A_ε^* algorithm.

The greedy heuristic is a two-stage algorithm. The first stage determines the order of task execution/allocation, and the second stage chooses the best processor that completes each task in the sequence at the earliest possible time. The order of task execution is determined using the notion of a level of a node in the task graph. Intuitively, the level of a node i in the task graph is the critical path length from the terminal node of the task graph to node i. When the multi-processor is made up of processors with different apeeds we assume that every task on a path from the terminal node to node i is executed on the fastest processor. The execution order of the tasks is based on decreasing node levels on the premise that the tasks with longer paths (or larger level) should be completed as soon as possible [Hu, 1961; Kohler, 1975]. The results of computational experiments on several hundred random graphs, the weapon-target assignment, and the multi-target tracking algorithms have shown that the greedy heuristic provides optimal mapping in over 75% of the test cases, but can be in error by as much as 230% from the optimal completion time in some test cases.

In order to improve the performance of the greedy heuristic, we developed a second mapping algorithm using the concept of pair-wise exchange. In effect, the algorithm exchanges the execution order of all possible pairs of tasks in the sequence, while satisfying the precedence relations of the task graph. This algorithm has been found to be optimal in over 85% of the test cases examined, and the average error in completion time was less than 5% of the optimal for all the test cases. This is an efficient and useful algorithm for large scale mapping problems, wherein an optimal solution is hard or impossible to find in polynomial time.

The performance evaluation of the heuristic and pair-wise exchange algorithms is accomplished using the optimal mapping algorithm, based on the heuristic A^* algorithm as a benchmark [Nilsson, 1980; Pearl, 1984]. The heuristic evaluation function (HEF) required by the A^* algorithm is the level of each node on the task graph. This HEF can be shown to be admissible, i.e., it is a lower bound on the optimal cost-to-go, which ensures that the A^* algorithm provides an optimal mapping. However, for large problems, the computational requirements of the A^* algorithm are prohibitive.

In order to subdue the combinatorial explosion of the A^* algorithm on large scale mapping problems, we developed an A^*_ε algorithm that guarantees that the completion time is within $(1+\varepsilon)$ of the optimal completion time. The tradeoff between computational complexity and optimality of the A^*_ε algorithm is controlled by the choice of ε.

1.4 MAPPER SOFTWARE PACKAGE

The mapping algorithms have been integrated into an interactive computer software package, termed MAPPER, for the analysis and performance evaluation of alternative BM/C^3 algorithms and multi-processor architectures. A functional structure of MAPPER is shown in Fig. 1-3. It consists of four functional modules: a graphical user interface, data translator, algorithm driver, and the algorithms. The user interface provides a mouse-driven graphical environment for users to draw and edit task, processor graphs, and enter their parameters. The results of MAPPER are displayed in the form of a Gantt chart, along with other useful performance measures like the speedup and the utilization of each processor. The data translator portion of MAPPER converts the user's view of task and processor graphs into algorithm-level inputs. The algorithm driver invokes the appropriate algorithm, based on the user's mouse-driven commands.

Finally, the algorithm portion of the MAPPER consists of the four mapping algorithms discussed earlier: greedy heuristic, pair-wise exchange, A^*, and A^*_ϵ algorithms. The MAPPER software is hosted on a SUN workstation.

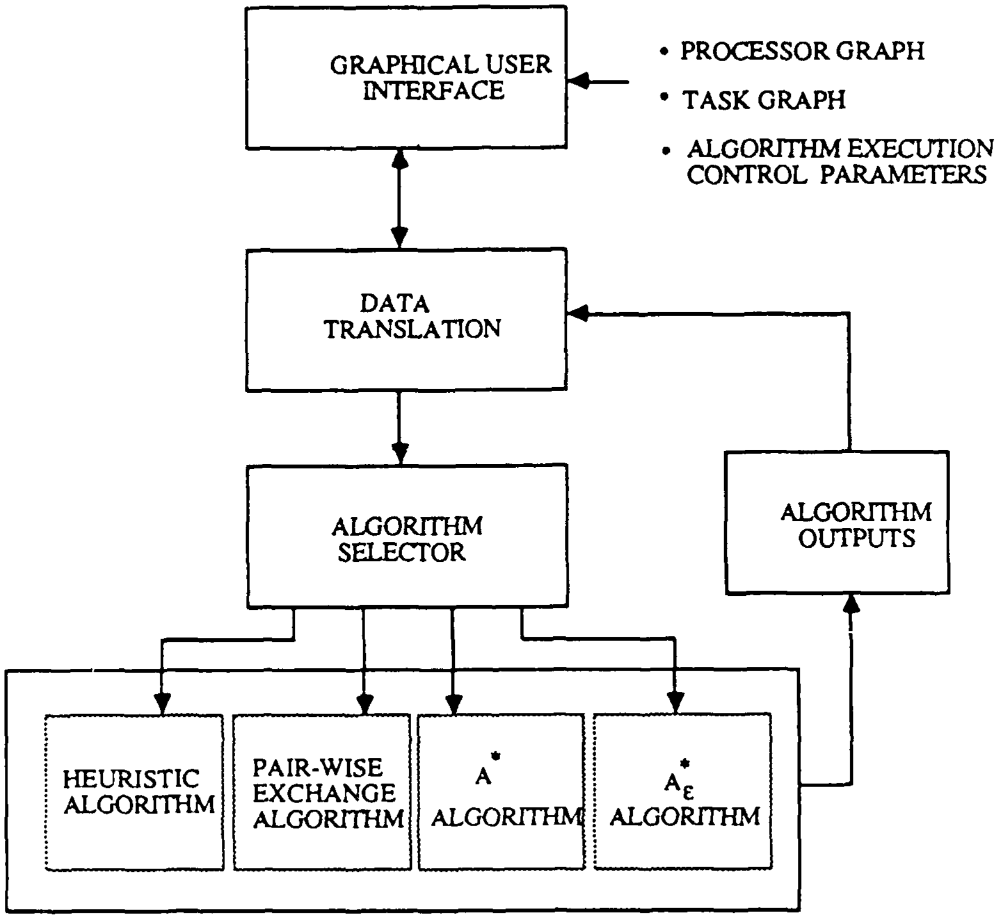


FIGURE 1-3 : A SOFTWARE ENVIRONMENT FOR ALGORITHM-ARCHITECTURE MAPPING

1.5 ORGANIZATION OF THE REPORT

In section 2, we provide a mathematical formulation of the static, deterministic mapping problem, and discuss its relation to previous scheduling techniques. Specifically, we show that simplified formulations lead to the following problems: (1) scheduling independent tasks on parallel identical processors, (2) scheduling tree-structured task graphs on parallel identical processors wherein each task requires unit processing time and zero communication delay, and (3) all previous mapping formulations discussed in the literature. Also included in this section are the drawbacks of the previous approaches, and the key features that distinguish our formulation from those of the earlier approaches.

In section 3, we derive the key mapping equation that forms the basis of all the four mapping algorithms developed in this report. We also derive the four algorithms and illustrate their performance on a simple example.

Section 4 provides four sets of computational experiments to demonstrate the performance of the mapping algorithms. The first set considers hypothetical examples gleaned from the literature. The second set of experiments, which deal with several hundred random graphs over a wide range of computation/ communication ratios, and were used to critically assess the performance of the heuristic and pair-wise exchange algorithms. The third and fourth set of experiments are related to the weapon-target assignment and the multi-target tracking algorithms.

Section 5 provides a summary of the research accomplishments and future research plans. Appendix A describes the design methodology for the user interface of MAPPER software package. Finally, Appendix B contains a user manual for MAPPER.

SECTION 2
STATIC MAPPING PROBLEM
FORMULATION AND PREVIOUS APPROACHES

2.1 PROBLEM FORMULATION

We formulate the problem of mapping a static task graph onto a static processor graph as one of minimizing the completion time of the task graph subject to (1) constraints on the memory available at each processor, (2) the replication level of each task, and (3) security. The task graph (also termed a problem graph or computation flow graph (CFG)) is a directed, acyclic graph, $G_t = (V_t, E_t)$, where $V_t = \{i : i = 1, 2, \cdots, N\}$ is the set of vertices (nodes) denoting the tasks of the application algorithm, and $E_t = \{ <i,j> : i,j = 1, 2, ..., N ; i \neq j \}$ is the set of *directed* edges (arcs, links) representing the inter-task communication between pairs of tasks i and j, and the partially ordered constraint that task i must precede task j. Each node i of the task graph is parameterized by the 3-tuple (s_i, m_i, r_i), where s_i is the service demand of task i measured in terms of millions of instructions, m_i is the memory requirement of task i measured in Kilo bytes (Kb) and r_i is the replication level of task i for fault-tolerance. Each directed edge $<i,j>$ of the task graph is parameterized by v_{ij}, where v_{ij} denotes the amount of data transmitted between tasks i and j measured in terms of bits. We assume, without loss of generality, that the task graph G_t consists of a start (source) node and a terminal (sink) node, i.e., the task graph G_t is such that each node can be reached if we go forward from the start node or go backward from the terminal node. If the task graph does not have a start node and/or a terminal node, a dummy start node/terminal node can always be added to the task graph such that the number of instructions of the dummy node and the data transmitted from the dummy node to other nodes of the task graph (and vice versa) are zero. For notational convenience, we assume that the terminal task corresponds to node N of the task graph, G_t.

In the same vein, the processor graph is an undirected graph, $G_p = (V_p, E_p)$, where $V_p = \{ p : p = 1, 2, ..., M \}$ is the set of vertices denoting the processors, and $E_p = \{ (p, q) : p, q = 1, 2, ..., M; p \neq q \}$ is the set of *undirected* edges depicting the communication links among processors. Each node of the processor graph is parameterized by the 2-tuple (μ_p, R_p), where μ_p is the service rate of processor measured in terms of millions of instructions per second, and R_p is the memory capacity of processor p in Kb. Each edge (p,q) of the processor graph G_p is parameterized by the link capacity, c_{pq} measured in bits per second. We assume that the data communication between a pair of processors (k,q) follows the shortest path $(k, k_1, k_2, \cdots, k_n, q)$ where $k, k_1, k_2, \cdots, k_n, q$ are the nodes on the shortest path. We assume that the task sequencing at each processor is nonpreemptive. Furthermore, we assume that each node of the processor graph contains an execution processor and a communication processor so that task execution and data communication can be serviced simultaneously at the same node. The reliability constraint is modeled as a redundant execution of each task i at r_i (≥ 1) distinct processors. Finally, the security constraint implies that each task i can be executed only at a certain set of distinct processors, S_i, $i \in V_t$. Clearly, $|S_i| \geq r_i$ for all $i \in V_t$, where $|S_i|$ is the cardinality of S_i. That is, the number of distinct processors to which a task i can be allocated must at least equal the replication level, r_i.

Note that in the case without any constraint, i.e., no memory, security, and storage constraints, the node parameter of the task graph can be represented by the service demand of task i, s_i, while the node parameter of the processor graph can be represented by the service rate of processor q, μ_q.

Formally, a mapping is a partition of the task set $V_t = \{1, 2, ...N\}$ into M ordered sets $T_1, T_2, ..., T_M$:

$$T_q = \left\{ q_1, q_2, \cdots, q_k \right\} k \geq 0, \qquad (2\text{-}1a)$$

$$\text{and} \bigcup_{1 \leq q \leq M} T_q = \left\{ 1, 2, ..., N \right\}, \qquad (2\text{-}1b)$$

that satisfy the precedence relationships of the task graph. Note that the ordered sets $T_1, T_2, \cdots, T_N$ are not disjoint due to redundant execution of tasks at multiple processors. We find it convenient to define the complementary distinct element sets P_i via $P_i = \{ q : i \in T_q \text{ and } \textit{no element is repeated} \}, \; i \in V_t$. That is, P_i is the set of distinct processors to which task i is allocated. In addition, let β_i denote the set of immediate parents of task i in the task graph, G_t. Then the static, deterministic mapping problem can be stated succinctly as follows:

$$\min_{\left\{ T_1, T_2, \cdots, T_M \right\}} CT_N \,(T_1, T_2, \cdots, T_M), \tag{2-2}$$

subject to:

$$\sum_{i \in T_q} m_i \le R_q \; ; \; q \in V_p \; (\textit{memory constraint}), \tag{2-3}$$

$$| \, P_i \, | = r_i \; ; \; i \in V_t \; (\textit{reliability constraint}), \tag{2-4}$$

$$\text{and } P_i \subset S_i \, , \, i \in V_t \; (\textit{security constraint}) \tag{2-5}$$

where $CT_N \,(T_1, T_2, \cdots, T_M)$ is the completion time of the terminal task N (or make span) under the mapping $(T_1, T_2, \cdots, T_M)$. Note that the mapping must satisfy the precedence constraints, viz., task i cannot begin executing on processor q, $q \in P_i$, until the data from each task of the parent set β_i is available at processor q, $i \in V_t$. We will provide a precise mathematical characterization of the precedence constraint in section 3.1.

Once the mapping $(T_1, T_2, \cdots, T_M)$ is known, the speedup can be obtained via :

$$\textit{speedup} = [\sum_{i \in V_t} r_i s_i \,] \, / \, [\mu_f \; CT_N(T_1, T_2, \cdots, T_M)] \, , \tag{2-6}$$

where μ_f is the service rate of the fastest processor. The processor utilization, U_q, is obtained from :

$$U_q = [\sum_{i \in T_q} \frac{s_i}{\mu_q}] / \; CT_N(T_1, T_2, \cdots, T_M) \, . \tag{2-7}$$

The utilization of links can be derived in a similar manner.

Illustrative Example:

To illustrate the problem formulation, consider the task and processor graphs shown in Fig. 2-1. The task graph is represented by $V_t = \{1, 2, 3, 4\}$, $E_t = \{ <1,2>, <1,3>, <2,4>, <3,4> \}$. The node parameters are: $(s_1, m_1, r_1) = (1, 5, 1)$, $(s_2, m_2, r_2) = (3, 10, 2)$, $(s_3, m_3, r_3) = (4, 20, 1)$, and $(s_4, m_4, r_4) = (2, 10, 1)$. The edge parameters of the task graph are: v_{12}=1, v_{13}=2, v_{24}=2, and v_{34}=1. Similarly, the processor graph is represented by: $V_p = \{1, 2\}$ and $E_p = \{ (1,2) \}$. The node parameters of the processor graph are: (μ_1, R_1)= (1,100) and (μ_2, R_2)= (1, 200). Since there is a single link, the edge parameter is: c_{12}=1. Let the security constraint be S_1={1,2}, S_2={1,2}, S_3={2}, and S_4={1,2}. Then T_1 = {1,2} and T_2 = {3,2,4} is a feasible mapping. With this mapping, we have P_1={1}, P_2={1,2}, P_3={2}, P_4={2}, and the completion time $CT_4(T_1,T_2)$ =12. The speedup for this mapping is, SP=1.08. The utilizations of the processors are: U_1= 0.33 and U_2 = 0.75.

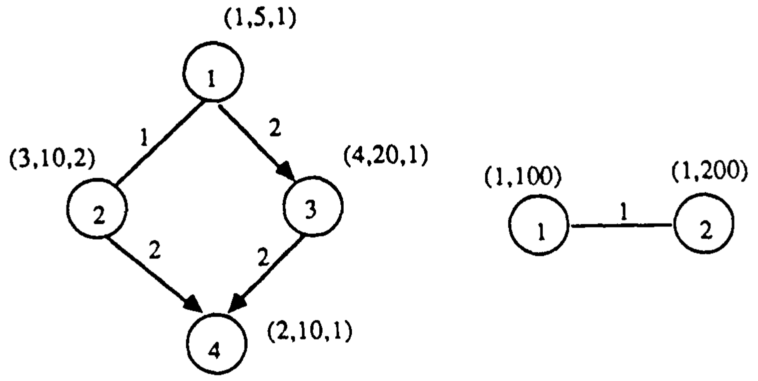


FIGURE 2-1: AN ILLUSTRATIVE EXAMPLE OF TASK AND PROCESSOR GRAPHS

To illustrate the problem complexity, consider the case where there are no security or memory constraints, i.e., each task can be executed on any processor. Then the total number of different allocations for task i is $\prod_{l=0}^{r_i-1} (M-l)$. The execution order of the tasks is upper-bounded by $N!$. Therefore, the total number of different mappings in the worst case is $N! \prod_{i=0}^{N} [\prod_{l=0}^{r_i-1} (M-l)]$. When $r_i=1$, for $1 \leq i \leq N$, the total number of different mapping in the worst case is $N!M^N$.

Indeed, the mapping problem posed above is NP-complete [Horowitz and Sahni, 1978; Garey and Johnson, 1979; Bokhari, 1979], which means that an optimal algorithm for the static, deterministic mapping problem with a run-time bound that is a polynomial function of tasks and the number of processors exists if, and only if, all combinatorial optimization problems, including the traveling salesman, maximum clique, and the satisfiability problems can be solved in polynomial time [Cook, 1971; Karp, 1972]. The evidence indicates that in all likelihood any problem which is NP-complete cannot be solved by an algorithm of polynomial time complexity. Therefore, all practical algorithms exploit the use of heuristics to reduce the computational burden. In the following subsection, we discuss the relationship of the mapping problem with previous scheduling problems in the literature in order to indicate its generality, and to provide a basis for developing the heuristic and optimal mapping algorithms of section 3.

2.2 RELATION TO PREVIOUS SCHEDULING TECHNIQUES

2.2.1 SCHEDULING INDEPENDENT TASKS

This problem is concerned with nonpreemptive scheduling of N independent tasks on M processors. Thus, we assume that the task graph is a vertex graph (i.e., a graph with isolated vertices and no edges) with no constraints on memory, reliability and security. The execution time of task i on processor j will be denoted by t_{ij} $(=\frac{s_i}{\mu_j})$ yielding an $N \times M$ nonnegative matrix of processing times. Since there are no reliability constraints, a schedule for M processors is a partition of the task set V_t = $\{ 1, 2, 3, \ldots, N \}$ into M disjoint ordered sets $T_1, T_2, \ldots, T_M$. It is shown that, for $M \geq 2$, the problem of obtaining a schedule to minimize the makespan (completion time) or, equivalently, tc maximize the speedup, is NP-complete [Horowitz and Sahni, 1978; Gary and Johnson, 1979]. This puts added importance on the development of heuristic algorithms.

An example of a heuristic algorithm for scheduling *independent* tasks on identical processors (i.e., $\mu_j = \mu$, $j = 1, 2, \ldots, M$) is the list scheduling (LS) strategy. The LS strategy schedules tasks according to a given priority list of tasks, and at each step the first available task on the list is assigned to a processor with the currently earliest completion time. A natural question then is: how good are the solutions generated by the suboptimal LS algorithm? Such a question is usually answered by assessing the worst case accuracy of the suboptimal algorithm. It has been proved [Graham, 1966; Syslo, Deo, and Kowalik, 1983] that the list scheduling algorithm generates a solution which satisfies:

$$C_{max}(LS) \le \left(2 - \frac{1}{M}\right) C^*_{max} , \tag{2-8}$$

where $C_{max}(LS)$ is the completion time with the LS strategy, and C^*_{max} is the optimal completion time. Thus, the completion time of the LS strategy can be worse by at most a factor of two than the optimal completion time.

Another example is to form a priority list according to nonincreasing order of processing times t_i. The list scheduling algorithm applied to such an ordering is called the largest processing time (LPT) algorithm. The LPT algorithm generates a solution which satisfies:

$$C_{max}(LPT) \le \left(\frac{4}{3} - \frac{1}{3M}\right) C^*_{max} . \tag{2-9}$$

This means that the completion time of an LPT schedule is at most 33% worse than that of the optimal solution.

In spite of these worst-case bounds, LPT rule behaves very well on random problems. In one experiment [Coffman, Garey, and Johnson, 1978], thirty tasks, with task times chosen according to a uniform distribution between 0 and 1, were generated. C^*_{max} was estimated to be $\frac{\sum t_i}{10}$, and $C_{max}(LPT)$ is the length of the LPT schedule generated. The experiment was repeated ten times and the values of of the relative approximation error, $\frac{C_{max}(LPT) - C^*_{max}}{C^*_{max}}$, were computed. The average error was found to be 0.074. In a second experiment [Coffman, Garey, and Johnson, 1978], task times were chosen according to a normal distribution. The average of the value, $\frac{C_{max}(LPT) - C^*_{max}}{C^*_{max}}$, was found to be 0.023. Indeed, it has been proved that if the processing requirements of the tasks are identically distributed random variables, then as the number of tasks, N, approaches infinity, the relative approximation error approaches zero (asymptotic optimality) with probability 1 [Loululo, 1984; Bruno and Downey, 1986; Frenk and Rinnooy Kan, 1987].

2.2.2 SCHEDULING TREE TASK STRUCTURES

From the preceding section, it is clear that the problem of minimizing makespan on M (where M ≥ 2) processors belongs to a class of difficult combinatorial problems. Nevertheless, in some cases, augmenting the problem with some restrictions converts it into a simplified problem. Obviously, the problem will be trivial if all tasks have unit processing time: the LS strategy is optimal. By introducing the precedence relations among tasks, nontrivial problems can be formulated. Suppose we have N unit processing time tasks with precedence relations in the form of a tree, which must be scheduled on M identical processors. Hu [1961] and Sethi [1976] proposed an O(N) optimal algorithm to solve this type of problem. Hu's idea was based on a LS strategy, where the priority list is formed based on node levels. The level of a node i in the task tree is defined as the number of nodes (including node i) on the path to the terminal node. A priority list is then constructed according to nonincreasing node levels, and then the LS scheduling algorithm is applied using the priority list.

For example, Fig. 2-2 shows a tree task structure with unit processing time on each task. Task a should be assigned first since it has the highest level (i.e., 4). The next task to be assigned is task c with its level equal to 3, and so on. Fig. 2-3 shows the results of scheduling this tree task structure on a two-processor system. Since the tasks are arranged in nonincreasing levels, we can also apply Hu's algorithm to solve problems (with nonidentical task processing times and directed acyclic task graphs) after minor modifications [Kohler, 1975]. The modification pertains to the definition of a node level. In this case, the level of a node i is the length of the longest path from node i to the terminal node, where the length of the path is measured in terms of the processing time . Therefore, one can say that the task to be scheduled next is the one that heads the current longest (critical) path in the precedence graph. Scheduling according to this rule is termed the critical path (CP) scheduling: It is list scheduling applied to a list of tasks arranged in nonincreasing longest paths.

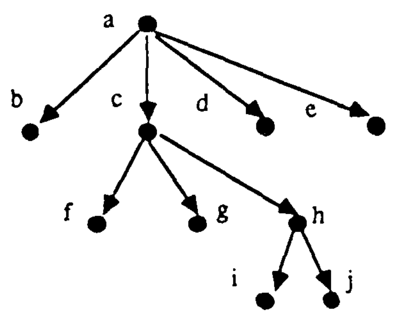
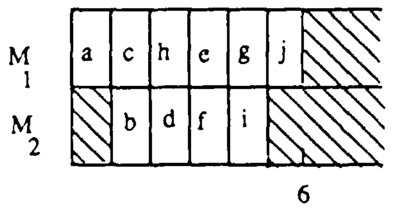


FIGURE 2-2: A TREE-STRUCTURED TASK SYSTEM



COMPLETION TIME = 6

FIGURE 2-3: LEVEL SCHEDULEING ON TWO PROCESSORS

2.2.3 PREVIOUS MAPPING APPROACHES

The mapping problem is similar to the previous scheduling problems except that precedence relationships may exist among tasks. In addition, the data dependencies among tasks induce a communication pattern among processors over limited capacity channels (links). Several approaches with different optimality criteria have been suggested to solve this type of problem: (1) network flows, (2) integer programming, (3) branch-and-bound, and (4) heuristic methods.

1. Network Flows:

Stone [1977] studied a two-processor mapping problem assuming no precedence restrictions on task execution. He used a modified intertask-communication graph by adding the processor nodes to the task graph such that an edge between a processor and each task of the original task graph. The weight on the new edge between a task and a processor is the processing cost of the task on the other processor. A cut of the modified graph separates the graph into two disconnected parts, each part representing an allocation of tasks to a processor. The cost of the allocation is equal to the sum of the weights on the cut and the minimum cut corresponds to the optimal mapping. Stone used the modified Ford-Fulkerson maxflow-mincut algorithm [Ford and Fulkerson, 1962] to minimize the sum of processing and communication costs. Since the precedence relationships among tasks are neglected in his model, the mapping $(T_1, T_2, \cdots, T_M)$ can be represented by the binary matrix X=$[x_{ik}]$ where $x_{ik}=1$, if task i is allocated to processor k and $x_{ik}=0$ otherwise. The cost function is then formulated as:

$$cost(X) = \sum_{k=1}^{2} \sum_{i=1}^{N} \left\{ f_{ik}\, x_{ik} + \sum_{l<k} \sum_{j<i} c_{ij}\, x_{ik}\, x_{jl} \right\}, \tag{2-10}$$

where f_{ik} represents the processing cost for task i on processor k, and c_{ij} is the communication cost between tasks i and j, when those tasks are assigned to different processors. The first summation represents the processing costs, while the second term represents the interprocessor communication cost between two-processors. Although the approach is very elegant, it has several shortcomings. First, the approach is limited to two processor systems. For a system where the number of processors is greater than two, the computational complexity soon becomes intractable. The second limitation of the approach is the difficulty in incorporating precedence relationships among tasks, and various constraints such as memory and redundancy into this model.

2. Integer Programming Approach

This method formulates the mapping problem as a binary 0-1 integer programming problem. The processing costs of tasks are represented by an $N \times M$ matrix F where f_{ij} represents the processing cost when task i is allocated to processor j. The interprocessor communication cost can be represented by a product of volume and the distance. If we define V=$\{v_{ij}\}$, $i = 1, 2,..., N$, $j = 1, 2,..., N$, where v_{ij} represents the amount of data to be transferred between tasks i and j, and C=$\{c_{kl}\}$, $k = 1, 2,.. M$ and $l = 1, 2,..., M$, is the distance matrix where c_{kl} is a measure of communication cost between processors k and l, then the communication cost between a task i allocated to processor k and a task j allocated to processor l, is $v_{ij}c_{kl}$. Neglecting the precedence constraints, we can formulate the objective function in terms of the allocation (assignment) matrix $X = [\, x_{ik} \,]$ as:

$$cost(X) = \sum_{k=1}^{M} \sum_{i=1}^{N} \left\{ f_{ik}\ x_{ik} + \sum_{l<k} \sum_{j<i} w\ v_{ij} c_{kl}\ x_{ik}\ x_{jl} \right\}. \tag{2-11}$$

The first summation term represents the processing cost for each task on its assigned processor. The second term represents the sum of interprocessor communication cost. The normalization constant w is used to scale processing and interprocessor communication costs and to account for any differences in units. In this approach, constraints on memory and real-time processing can be easily added. A limited memory environment can be represented by

$$\sum_{i=1}^{N} m_i x_{ik} \le R_k\ , \quad k=1,2,...,M\ , \tag{2-12}$$

where m_i represents the amount of memory required by task i and R_k represents the memory capacity at processor k. The real-time constraint can be represented by

$$\sum_{i=1}^{N} t_{ik} x_{ik} \le d_k \ , \quad k=1,2,\ldots,M \ , \qquad (2\text{-}13)$$

where t_{ik} represents the processing time of task i on processor k and d_k represents the time limit for processing the tasks that reside in processor k. The equation states that the time required to complete all the tasks assigned to a processor must not exceed the time limit. The integer programming approach provides a good representation of the task allocation environment. The constraints can be easily added or changed in the model and is an appropriate representation of the application algorithms. The approach is usually limited to small size mapping problems, since the computational and memory requirements grow exponentially with the problem size. In addition, the approach ignores queueing delays at the communication links, i.e., whenever data communication needs to take place between two processors, a path between two processors is assumed to be free.

3. Branch-and-bound

Shen and Tsai [1985] employ a branch-and-bound method using the minimax criterion to select an optimal task assignment. This has been shown to be isomorphic to a graph matching problem termed weak homomorphism. The search for optimal weak homomorphism corresponding to an optimal task assignment is solved via a state space search. Due to weak homomorphism, neighboring tasks are always assigned to neighboring processors. Their approach is a heuristic search similar to A* algorithm and the objective function is more realistic than the previous ones. However, precedence relationships among tasks are neglected in their problem formulation.

4. Heuristic Method

Heuristic approaches provide fast and effective means to obtain suboptimal mapping solutions. Kasahara and Narita [1984] propose a heuristic method termed critical path/most immediate successors first (CP/MISF) to minimize the completion time of the terminal task. The level l_i of task i used in CP/MISF method is defined to be the longest path length from the terminal node to the node i. Their approach consists of the following three steps:

a. Determine the level of each task.

b. Form a priority list in nonincreasing order of node levels and the number of immediate successor tasks.

c. Employ LS strategy on the priority list.

The method of CP/MISF is an improved version of critical path scheduling. From the previous discussion on scheduling independent and tree structured tasks, the worst case performance of this algorithm is given by the following equation.

$$\frac{CT_{\max} - CT^{*}_{\max}}{CT^{*}_{\max}} \le 1 - \frac{1}{M} \qquad (2\text{-}14)$$

where $CT_{\max}$ is the completion time with CP/MISF method and $CT^{*}_{\max}$ is the optimal completion time. After the CP/MISF method is applied to find a solution, another method, termed depth first/immediate heuristic search (DF/IHS), is then employed to improve the CP/MISF method by using the completion time of CP/MISF method as an upper bound. In their approach, communication overhead among tasks is neglected.

Bokhari [1981A] formulated the mapping problem as one of maximizing the cardinality of mapping, i.e., the number of edges of the task graph that fall on the links in the processor graph. The heuristic algorithm proposed by Bokhari involves the iterative interchange of mapped nodes to increase the cardinality of mapping at each iteration. There are several drawbacks to this approach. The processors and links are assumed to be identical. In addition, the computation is assumed to be regular, i.e., all tasks have identical processing times and the amount of data to be transmitted between a pair of tasks is the same. Later Bokhari [1981B] employed a dynamic programming method to minimize the sum of execution and interprocessor communication cost for tree task structures. If the cost function involves time, this approach will minimize the serial execution time, i.e., at most only one processor is active at any time.

Lee and Agarwal [1987] suggest a new objective function which takes the weight of the task edges and the actual system distance into account to minimize the communication overhead. The basic idea of their mapping strategy is that tasks communicating more frequently than the others are to be placed closer. In their problem formulation, the objective function can quantify the real communication among tasks more accurately, but the computation time of tasks is neglected. Furthermore, a system node can accommodate at most one task, which means that the number of processors should be greater than or equal to the number of tasks.

2.3 DRAWBACKS OF THE PREVIOUS MAPPING APPROACHES

The previous mapping approaches, although varied in terms of solution techniques used, have two major shortcoming. First, the optimality criterion involving the sum of processing and communication costs neglect the synchronization delays due to precedence constraints, and assume that the communication delay in transferring data between a pair of processors is independent of the data traffic between the processors. In queueing parlance, the constant delay assumption is tantamount to modeling each communication link by an infinitive server [Lavenberg, 1983]. Second, the approaches that take into account the precedence constraints assume zero communication delays [Kasahara and Narita, 1984], and those that consider communication delays neglect synchronization delays [Shen and Tsai, 1985].

The mapping algorithms developed in this report overcome the drawbacks of previous approaches. The key features of our problem formulation include: (1) explicit consideration of precedence restrictions among tasks and, hence, the synchronization delays, (2) sequencing of data messages to account for queueing delays at communication links, and (3) the incorporation of storage, security, and fault-tolerance requirements. In the next section we develop both heuristic and optimal algorithms to solve the general mapping problem formulated in section 2.1.

SECTION 3

MAPPING ALGORITHMS

3.1 KEY MAPPING EQUATION

In the general mapping problem formulated in subsection 2.1, the completion time of a task i on an assigned processor q is a function of: (1) the service demand of task i and the service rate of processor q; (2) the time at which the data from immediate predecessor (or parent) tasks of task i is available at processor q; and (3) the time when processor q becomes available. The delay due to data dependency accounts for synchronization requirements and communication delays. The consideration of processor availability serves as a model for queueing delays due to previously assigned tasks. In this subsection, we derive an explicit equation for the evolution of the completion time as a function of mapping, which forms the basis for our mapping algorithms of sections 3.2-3.4. For ease of exposition, we derive the mapping equation for the unconstrained and constrained problems separately. The mathematical notation used in this section is shown in table 3-1.

3.1.1 Mapping Equation without Constraints

Consider the mapping problem without memory, security, and redundancy constraints. Let $(T_1, T_2, ..., T_q, ..., T_M)$ denote a partial mapping and let $CT_j\ (T_1, T_2, ..., T_M)$ be the corresponding completion time of task j, $j \varepsilon V_t$. Since there are no redundancy constraints, the ordered sets T are disjoint. Suppose we want to assign a ready task i to processor q so that the new mapping is $(T_i, T_2, ..., T_q \cup i, ..., T_M)$. We would like to derive an expression for the completion time of task i on processor q, $CT_j\ (T_1, T_2, ..., T_q \cup i, ..., T_M)$, given the partial mapping and the corresponding completion times.

term	meaning
V_t	set of tasks in the task graph
V_s	set of processors in the processor graph
M	number of processors = $\mid V_s \mid$
N	number of tasks = $\mid V_t \mid$
s_i	number of instructions of task i in millions
m_i	memory requirement of task i in Kb
r_i	redundancy of task i
v_{ij}	amount of data transferred between tasks i and j in K bytes
β_i	set of immediate predecessor (parent) tasks of task i
α_i	set of immediate successors of task i
μ_q	service rate of processor q in millions of instructions per second (MIPS)
R_q	memory capacity of processor q in Kb
c_{pq}	capacity of link (p,q)
S_i	set of processors where task i can be assigned due to security constraints, $i \; \varepsilon V_t$
T_q	set of tasks assigned to processor q, $q \; \varepsilon V_s$
P_i	set of distinct processors to which task i is assigned, $i \; \varepsilon V_t$
$CT_{i,a}$	completion time of the a^{th} copy of task i, a=1, 2, ..., r_i; $i \varepsilon V_t$
CT_i	completion time of task i
l	last task assigned to processor q
A_{pq}	the available time of link (p,q)
$i[y_j,a]$	j^{th} earliest completed parent of task i. task y_j is the corresponding parent task, and a is the a^{th} copy of y_j, $y_j \varepsilon \beta_i$ and a = 1, 2, ..., r_{y_j}, $i \; \varepsilon \; V_t$
$B^q_{i[y_j,a]}$	the time at which data from the j^{th} earliest parent of task i, denoted by $i[y_j, a]$, is available at processor q, i ε V_t, q ε V_s
$i[j]$	j^{th} earliest completed parent of task i, j = 1, 2, ..., $\mid \beta_i \mid$ when redundancy r_i = 1, $i \; \varepsilon \; V_t$
$B^q_{i[j]}$	the time at which data from the j^{th} earliest completed parent of task i is available at processor q when redundancy r_i=1, $i \; \varepsilon \; V_t$, $q \; \varepsilon \; V_s$
M_i	set of processors to which task i can be assigned without violating the memory constraint
D^q_i	latest time at which data from all the parents of task i is available at processor q

TABLE 3-1: DEFINITION OF VARIABLES

As discussed earlier, task i can not begin execution on processor q until the data from predecessors of task i is available at processor q, and until processor q completes the execution of the end task in the ordered set, T_q. Let D^q_i denote the latest time at which data from all the predecessors of task i becomes available at processor q, and let l be the end task in the ordered set, T_q. Then, the start time of task i is max[D^q_i, CT_l ($T_1, T_2, ..., T_q, ..., T_M$)] and the completion time is given by

$$CT_i \; (T_1, T_2, ..., T_q \cup i, ..., T_M) = \max \; [\; D^q_i \; , CT_l \; (T_1, T_2, ..., T_q, ..., T_M \;) \;] + \frac{s_i}{\mu_q} \qquad (3\text{-}1)$$

The latest time at which data from all the predecessors of task i becomes available at processor q is:

$$D_i^q = \max_{j=1,2,\ldots,|\beta_i|} B_{i[j]}^q \tag{3-2}$$

where $B_{i[j]}^q$ is the time at which data from the j^{th} earliest completed parent task of task i becomes available at processor q and β_i is the set of parents of task i. To compute $B_{i[j]}^q$, we make the reasonable assumption that the data communication from parent tasks takes place in the order in which they are completed, i.e., the first completed parent task sends its data first. The variables $B_{i[j]}^q$ can be computed via the following algorithm.

Algorithm 3.1: *Computation of* $B_{i[j]}^q$. Given the completion times of parent tasks of a task i, $CT_{i[j]}$, $1 \le j \le |\beta_i|$ arranged in nonincreasing order, and the link available times A_{pq}, the following algorithm computes $B_{i[j]}^q$.

```
For $j$ = 1 to $|\beta_i|$ do
    $k_0 := \{ p : i[j] \in T_p \}$
    If $k_0 \neq q$ then
        Find the shortest path $(k_0, k_1, ..k_n, ..q)$ from processor $k_0$ to processor $q$
        $A_{k_0 k_1} := \max \{ A_{k_0 k_1}, CT_{i[j]} \} + \frac{V_{i[j]i}}{c_{k_0 k_1}}$
        $k_{n+1} := q$
        For $i$ = 1 to n do
            $A_{k_i k_{i+1}} := \max ( A_{k_{i-1} k_i}, A_{k_i k_{i+1}} ) + \frac{V_{i[j]i}}{c_{k_i k_{i+1}}}$
        end do
        $B_{i[j]i}^q := A_{k_n k_{n+1}}$
    else
        $B_{i[j]}^q := CT_{i[j]}$
    end if
```

end do

3.1.2 Mapping Equation with Constraints

The memory and security constraints simply restrict the feasible processor assignments for a task. However, the redundancy constraint imposes additional synchronization delays, since a task i can not begin execution until the data from all copies of the parent tasks of task i is available at the assigned processor. Therefore, we must keep track of the completion time of each copy of a task. In addition, the ordered sets T_q in $(T_1, T_2,..., T_q, ..., T_M)$ are not disjoint, since a task i must be replicated at multiple processors.

To derive the mapping equation, let M_i denote the set of processors to which a ready task i can be assigned without violating the memory constraints, i.e.,

$$M_i = \left\{ q : m_i + \sum_{j \varepsilon T_q} m_j \leq R_q \right\} \qquad (3\text{-}3)$$

and let S_i be the set of processors to which task i can be assigned based on security considerations. Then the set of feasible processor assignments, F_p is given by

$$F_p = \left\{ q : q \varepsilon M_i \cap S_i \right\} \qquad (3\text{-}4)$$

The completion time of (any copy of) task i on processor q, $q \varepsilon F_p$ is given by:

$$CT_{i,a}(T_1, T_2,..., T_q \cup i, ..., T_M) = CT_i(T_1, T_2,..., T_q \cup i, ..., T_M)$$
$$= \max\left(D_i^q, CT_l(T_1, T_2,..., T_q, ..., T_M) \right) + \frac{s_i}{\mu_q} \qquad (3\text{-}5)$$

where D_i^q should be interpreted as the latest time at which data from all copies of the parent set β_i becomes available at processor q. If we let $i[y_j, a]$ denote the j^{th} earliest completed parent task of task i where y_j is the identity of the parent task and a is its copy $(1 \leq a \leq r_{y_j})$, then D_i^q is given by:

$$D_i^q = \max_{y_j \varepsilon \beta_i} \max_{1 \leq a \leq r_{y_j}} B_{i[y_j, a]}^q \qquad (3\text{-}6)$$

where $B_{i[y_j,a]}$ is the time at which data from copy a of task y_j becomes available at processor q. For a feasible assignment of task i on processor q, the variables $B^q_{i[y_j,a]}$ can be computed as follows:

Algorithm 3.2: *Computation of* $B^q_{i[y_j,a]}$. Given the completion times of each copy of parent tasks, $CT_{i[y_j,a]}$ arranged in decreasing order and the link available times, A_{pq}, the following algorithm computes $B^q_{i[y_j,a]}$.

```
For each $(y_j, a)$, $y_j \in \beta_i$ and $1 \le a \le r_{y_j}$ do
    $k_0$ := assigned processor of the $a^{th}$ copy of task $y_j$
    If $k_0 \neq q$ then
        Find the shortest path $(k_0, k_1, \cdots, k_n, q)$ from processor $k_0$ to processor $q$
        $A_{k_0k_1} := \max \left\{ A_{k_0k_1}, CT_{i[y_j,a]} \right\} + \frac{v_{y_ji}}{c_{k_0k_1}}$
        $k_{n+1} := q$
        For $i = 1$ to n do
            $A_{k_ik_{i+1}} := \max ( A_{k_{i-1}k_i}, A_{k_ik_{i+1}} ) + \frac{v_{y_ji}}{c_{k_ik_{i+1}}}$
        end do
        $B^q_{i[y_j,a]} := A_{k_nk_{n+1}}$
    else
        $B^q_{i[y_j,a]} := CT_{i[y_j,a]}$
    end if
end do
```

3.2 HEURISTIC MAPPING ALGORITHM

The heuristic algorithm consists of two-stages. The first stage employs the concept of critical path to determine the order of task execution, while the second stage sequentially allocates the tasks from the ordered list to processors so that the completion time of the tasks is a minimum. To determine the order of task execution, we define the level of a task i as:

$$l_i = \max_{k} \sum_{j \in \pi_k} \frac{s_j}{\mu_f} \tag{3-7}$$

where $\mu_f = \max_{q \in V_s} \mu_q$ is the service rate of the fastest processor, s_j is the service demand of task j, and π_k denotes the k^{th} path from task i to the terminal task. That is, l_i is the length of the critical path from task i to the terminal task. By construction, l_i is a lower bound on the completion time of a task graph rooted at task i. Following the level algorithm of Hu [1961] and Sethi [1976] discussed in section 2.2, the heuristic algorithm is based on the premise that tasks with larger levels should be executed earlier in the sequence. If several tasks have the same level, then the task with the greater number of successors should be completed first. Thus, we construct the execution order according to nonincreasing levels first, and nonincreasing successors next if all the tasks have the same levels.

Once the priority list is constructed, we sequentially allocate tasks to processors to minimize the completion time. In the case without constraints, task i is assigned to processor q^*, where

$$q^* = arg \min_{q} CT_i \; (T_1, T_2,..., T_q \cup i, ..., T_M) \tag{3-8a}$$

In the constrained mapping problem, we assign task i to r_i distinct processors $q_1^*, q_2^*, ..., q_{r_i}^*$ that yield minimum completion time. That is, the assignments $q_k^* (1 \le k \le r_i)$ are such that:

$$\begin{aligned} CT_i(T_1, T_2,..., T_{q_1^*} \cup i, ..., T_M) &\le CT_i(T_1, T_2,..., T_{q_2^*} \cup i, ..., T_M) \le .... \\ &\le CT_i(T_1, T_2,..., T_{q_{r_i}^*} \cup i, ..., T_M) \le .... \\ &\le CT_i(T_1, T_2,..., T_{q^*_{|F_p|}} \cup i, ..., T_M) \end{aligned} \tag{3-8b}$$

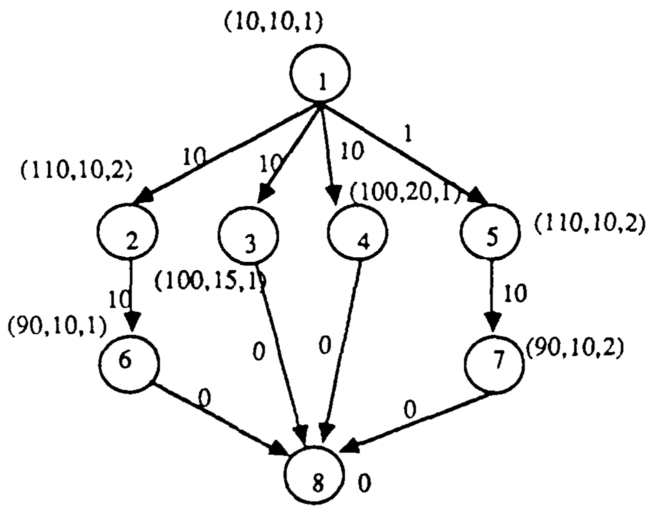
The heuristic mapping algorithm proceeds as follows:

Algorithm 3.3: *Heuristic Mapping Algorithm.* Given a directed task graph $G_t = (V_t , E_t)$ with parameters $(s_i , m_i , r_i , v_{ij})$ and the processor graph $G_s = (V_s , E_s)$ with parameters (μ_p , R_p , c_{pq}), the heuristic algorithm computes the mapping $(T_1, T_2, ..., T_q, ..., T_M)$, where T_q is the ordered set of tasks allocated to processor q.

```
Step 1: Determine the level of each task
        $Z = \{N\}$
        Repeat until  $Z = \phi$
            Select a task $i$ of $Z$ such that no successors of task $i$ appear in $Z$
            Compute the level of task $i$, $l_i$ via: $l_i = \max_{j \varepsilon \alpha_i} ( l_j ) + \frac{s_i}{\mu_f}$
            $Z = Z - \{i\} \cup \beta_i$
        end
Step 2: Construct a priority list [1] [2] ... [N] by sorting $l_i$ in nonincreasing order.
        Break ties on the basis of number of successors.
Step 3: For $j = 1$ to $N$ do
            $i = [j]$
            Form the feasible set of processors $F_p$ via Eqs. (3-3) and (3-4)
            Find assignments $q_k$ ( $1 \le k \le r_i$ ) via Eq. (3-8)
            $T_{q_k} = T_{q_k} \cup i$ ; $k=1, 2, ..., r_i$
            Update the link available times via Algorithm 3.1 or 3.2 as appropriate
        end do
```

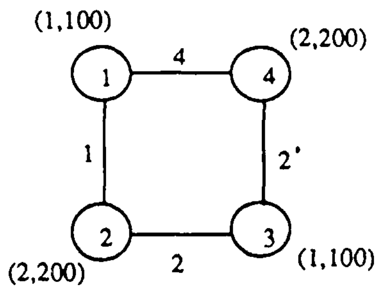
Illustrative Example:

Suppose the task and processor graphs are as shown in Fig. 3-1. We use step 1 to determine the levels of tasks, $l_1=105, l_2=100, l_3=50, l_4=50, l_5=100, l_6=45, l_7=45, l_8=0$. The priority list can be constructed by the nonincreasing levels, or [1]=1, [2]=2, [3]=5, [4]=4, [5]=3, [6]=6, [7]=7, and [8]=8. We construct the set of feasible processor assignments F_p for task [1]=1 and have $F_p = \{q : q \in M_1 \cup S_1\} = \{1, 2\}$. Since task [1]=1 can only be assigned to processors 1 and 2, we select processor 2 which has the earlier completion time for task 1. Similarly, the set of feasible processor assignments F_p for task [2] is $F_p = \{1, 2, 3, 4\}$. Since task [2]=2 has a replication level of 2, we assign task 2 to two different processors 2 and 4 with concomitant completion times of 60 and 70 time units. The set of feasible processor assignments F_p for task [3]=5 is $\{2, 3, 4\}$. Similarly, task [3]=5 has a replication level of 2, we assign task 5 to two different processors 2 and 1 with corresponding completion times of 115 and 116 time units. F_p for task [4]=4 is $\{1\}$ so we assign task [4]=4 to processor 1 due to the security constraint. Tasks [5] to [8] can be assigned in the same way. The completion time of the terminal task, task 8, is 216 for this example.



TASK GRAPH

task number	security matrix
1	1 1 0 0
2	1 1 1 1
3	0 1 1 1
4	1 0 0 0
5	1 1 1 1
6	1 1 1 1
7	1 1 0 1
8	1 1 1 1



PROCESSOR GRAPH

FIGURE 3-1: TASK AND PROCESSOR GRAPHS FOR AN ILLUSTRATIVE EXAMPLE

3.3 PAIR-WISE EXCHANGE ALGORITHM

The performance of the heuristic algorithm can be improved by iteratively exchanging the order of elements in the priority list, while satisfying the precedence constraints. The following lemma defines a feasible exchange of tasks in a priority list.

Lemma 3.1:

If [1] [2]..[j]..[k]..[N] is a priority list, then [1] [2]..[k]..[j]..[N] is also a feasible priority list only if all the successors of task [j] are executed after task [k] and all the parents of task [k] are executed before task [j], i.e., $\alpha_{[j]} \subset \{[k+1], [k+2], .., [N]\}$ and $\beta_{[k]} \subset \{[1], [2], .., [j-1]\}$.

Proof:

We will prove this by contradiction. Since [1] [2]..[j]..[k]..[N] is a feasible priority list, all the predecessors of task [j] must be executed before task [j] and all the successors of task [k] must be executed after task [k], i.e., $\beta_{[j]} \subset \{[1], [2], .., [j-1]\}$ and $\alpha_{[k]} \subset \{[k+1]\ [k+2]..[N]\}$. If any successor of task [j] or any predecessor of task [k] is executed between tasks [j] and [k], then an exchange on the order of execution of tasks [j] and [k] will require the execution of at least one task prior to its predecessor. That is, the new priority list will violate the precedence constraint, completeing the proof.

Note that the start and terminal tasks (i.e., tasks [1] and [N]) can not be exchanged with any task. The basic idea of pair-wise exchange is illustra.ed in Fig. 3-2. The pair-wise exchange algorithm considers all possible exchanges of the type illustrated in Fig. 3-2, and terminates when all feasible exchanges are exhausted. The algorithm proceeds as follows:

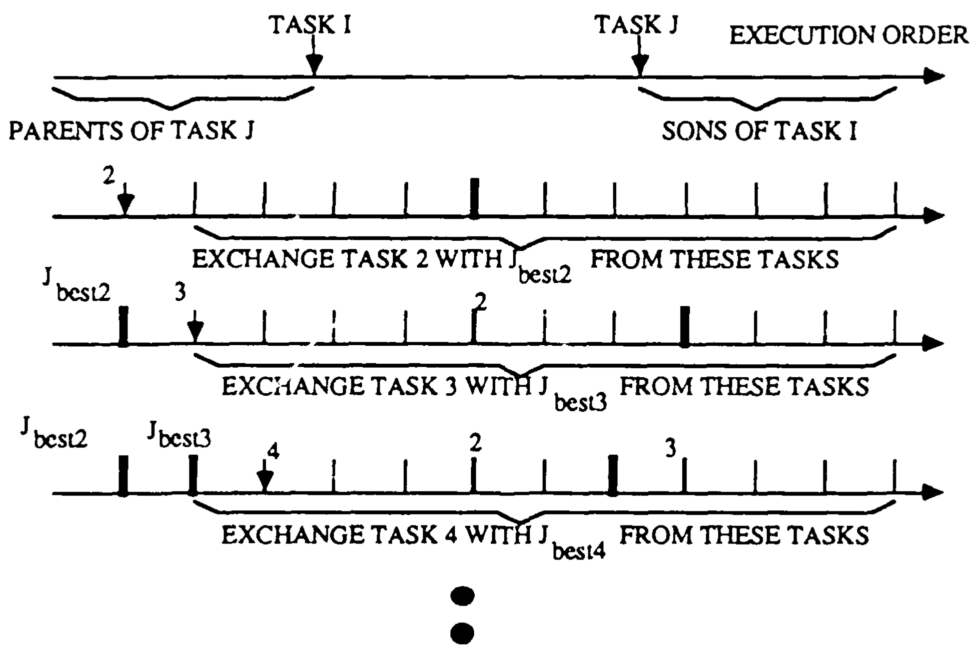

TASK I
TASK J
EXECUTION ORDER
PARENTS OF TASK J
SONS OF TASK I
2
EXCHANGE TASK 2 WITH J_{best2} FROM THESE TASKS
J_{best2}
3
2
EXCHANGE TASK 3 WITH J_{best3} FROM THESE TASKS
J_{best2}
J_{best3}
4
2
3
EXCHANGE TASK 4 WITH J_{best4} FROM THESE TASKS

FIGURE 3-2:: CONCEPT OF PAIR-WISE EXCHANGE ALGORITHM

Algorithm 3.4: *Pair-wise Exchange Algorithm.* Given a directed task graph $G_t = (V_t , E_t)$ with parameters $(s_i , m_i , r_i , v_{ij})$ and the processor graph $G_s = (V_s , E_s)$ with parameters (μ_p , R_p , c_{pq}), the pair-wise exchange algorithm computes the mapping $(T_1, T_2, ..., T_q, ..., T_M)$, where T_q is the ordered set of tasks allocated to processor q.

```
Step 1: Same as step 1 of algorithm 3.3
Step 2: Same as step 2 of algorithm 3.3
Step 3: For j=2 to N-2 do
            flag=0
            For k=j+1 to N-1 do
               If tasks [j] and [k] are exchangeable then
                  Initialize processor and link available times
                  Swap tasks [j] and [k] to construct a new priority list
                  Use step 3 of algorithm 3.3 to find a new mapping
                  If the result is better then
                     k':= k
                     flag=1
                     Update the result
                  end if
                  Swap tasks [j] and [k]
               end if
            end do
            If flag=1 then
               swap tasks [j] and [k']
            end if
         end do
```

In algorithm 3.4, the inner do loop finds the best task [k'] for the j^{th} position of the execution order and the outer do loop controls the value of j and then exchanges the j^{th} position with the best task found, if it results in a better mapping. Since there exists a start task and a terminal task in the task graph, it is obvious that [1']=[1] and [N']=[N] and hence no other tasks are exchangeable with these two tasks.

Illustrative Example:

Consider task and processor graphs shown in figure 3-1. We construct the priority list [1]=1, [2]=2, [3]=5, [4]=4, [5]=3, [6]=6, [7]=7, [8]=8. Using the heuristic algorithm, we have completion time 216. Now we exchange the priority list of tasks [2] and [3], the new priority list becomes [1]=1, [2]=5, [3]=2, [4]=4, [5]=3, [6]=6, [7]=7, [8]=8. Now use step 3 of algorithm 3.3 to find a new mapping with its completion time 205, so we keep this priority list and its corresponding mapping. We exchange the the other pairs of priority list such as tasks [2]=2 and [4]=4, [2]=2 and [5]=3 and find the completion time of the terminal task based on these priority lists is not earlier than 205. Hence, we swap tasks [2] and [3], fix task [2]=5, and get a new priority list [1]=1, [2]=5, [3]=2, [4]=4, [5]=3, [6]=6, [7]=7, [8]=8. We continue exchange tasks [3]=2 and [4]=4, [3]=2 and [5]=3, and so on. Since we cannot find an earlier completion time than 205, so the output will be the priority list [1]=1, [2]=5, [3]=2, [4]=4, [5]=3, [6]=6, [7]=7, [8]=8, the corresponding mapping, and the completion time 205.

3.4 OPTIMAL MAPPING (A^* *AND* A^*_ϵ) ALGORITHMS

Since the heuristic and pair-wise exchange algorithms do not guarantee an optimal solution, we develop an optimal mapping (A*) algorithm which forms a bench mark against which to evaluate the performance of two heuristic algorithms. The A* algorithm employs the heuristic algorithm to find an upper bound (UB) on the completion time and then searches for optimal allocation from all possible combinations of sequencing orders and allocations. The state space of the task allocation problem can be conceptualized as a decision tree, wherein node n is parameterized by the 5-tuple $(i_n, P_{i_n}, f_n, ,CT_{i_n}, \tau_n)$, where i_n is the task at node n, P_{i_n} is the set of processors where task i_n is assigned to, f_n is the cost constrained to go through node n, CT_{i_n} is the maximum completion time of task i_n at node n, and τ_n is the parent node of node n. The heuristic evaluation function (HEF), also termed cost selection function f_n, used by A* algorithm consists of three parts, g_n, h_n, and f_{τ_n}, where $g_n = CT_{i_n}$, h_n is the estimated cost of those unassigned tasks at node n, and f_{τ_n} is the cost of its parent node for node n. Since a task graph may consist of several routes starting from the start task to the terminal task, we may generate a node n with its task i_n lying in a path while the predecessor node τ_n with its task i_{τ_n} lying in other paths. Once we visit node n, we may use the information of HEF at node τ_n, f_{τ_n}, as a reference if the value of $g_n + h_n$ is less than f_{τ_n}. The reason is due to the lower bound cost of going through node τ_n is f_{τ_n}, so f_{τ_n} is also a lower bound cost at node n. As explained in section 3.1, the level l_i of task i is a lower bound on the completion time of a task graph rooted at task i. Since the processing time of task i_n on processor set P_{i_n} has been included into CT_{i_n}, the estimated cost of those unassigned tasks is equal to the level of task i_n, l_{i_n}, minus the service time of task i_n on the fastest processor, s_{i_n}/μ_f. In other words,

$$h_n = l_{i_n} - \frac{s_{i_n}}{\mu_f} \tag{3-9}$$

The heuristic evaluation function (HEF, also termed cost selection function) f_n is then determined by:

$$f_n = \max (g_n + h_n \, , f_{\tau_n}) \tag{3-10}$$

The HEF, f_n, defined in this way is admissible, i.e., a guarantee for an optimal solution. We start from the start node (root node, node 0) with its estimated cost h_0 evaluated to be:

$$h_0 = \max (l_{[1]} \, , [\sum_{i=1}^{N} s_i / \sum_{i=1}^{M} \mu_i]) \tag{3-11}$$

where $l_{[1]}$ is the level of the start task, $\sum_{i=1}^{N} s_i$ is the total service demands of tasks,

$\sum_{i=1}^{M} \mu_i$ is the service rate of the computer system. The estimated completion time is known to be lower bounded by the level of the start task. Since the total service demands of tasks, $\sum s_i$, divided by the service rate of the computer system, $\sum \mu_i$, is another lower bound of completion time, h_0 is taken from the larger of these two values. The cost at node 0 is $f_0 = g_0 + h_0 = h_0$, since $g_0 = 0$.

The ready task set at node 0 is the start task set {[1]} where [1] represent the start task. We construct the feasible processor assignment set F_p for task [1] and then expand node 0 by assigning the start task, task [1], to processor set $P_{[1]}$ for every $P_{[1]} \subset F_p$ and evaluate the corresponding 5-tuple $(i_n, P_{i_n}, f_n, ,CT_{i_n}, \tau_n)$ at each node n. If the value of estimated cost at node n, f_n, is less than or equal to the upper bound of completion obtained from the heuristic algorithm, we retain node n in the decision tree. Otherwise, node n is "fathomed". Once a node has been expanded, we put that node into a set CLOSED. We continue to select a node n for expansion whose f_n is a minimum among those unexpanded nodes. The ready task set at node n can be constructed from the path starting from node 0 to node n, $P_{0 \to n}$ and the precedence relationships of the task graph. Once the ready task set is constructed, each task (element) i in the ready task set at node n is again assigned to processor set P_i for every $P_i \subset F_p$. The number of new nodes generated by assigning tasks to processors is $\sum_{i \,\varepsilon\, ready\ task\ set} C_{r_i}^{|F_p|}$. Among these new nodes, only those tuples whose f_ζ is less than or equal to the upper bound (UP) need to be retained. We repeat this procedure until we construct an empty ready task set at node δ. Note that every node expansion will add new nodes to the decision tree. A complete path is the path starting from node 0 to the goal node δ. All the other paths are incomplete. The priority list on each processor, the completion of tasks and their corresponding allocations can be known from the optimal path, $\pi_{0 \to \delta}$. The completion time of the goal node, CT_{i_δ} is the completion time of the optimal mapping algorithm. Let T_q^m be the set of tasks assigned to processor q at node m, Z_m be a set contains the ready task set at node m, and σ_m corresponds to the processor at node m. The A* algorithm is described in the following:

Algorithm 3.6: *Optimal mapping (A*) algorithm.* Given a directed task graph $G_t = (V_t, E_t)$ with parameters (s_i, m_i, r_i, v_{ij}) and the processor graph $G_s = (V_s, E_s)$ with parameters (μ_p, R_p, c_{pq}), the A^* algorithm computes the optimal mapping $(T_1, T_2, .., T_q, ..., T_M)$, where T_q is the ordered set of tasks allocated to processor q.

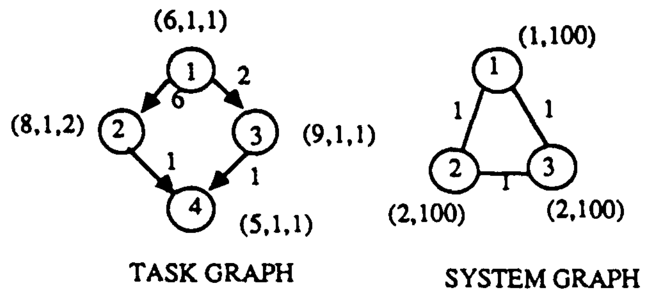
```
Step 1: Use algorithm 3.3 to find an upper bound (UB) on the completion time
Step 2: C=$\phi$
        n=0
        Z={[1]}
        Select a task $i$ of Z such that no predecessors of task $i$ appear in Z
        cost-to-go:= $l_i - \frac{s_i}{\mu_f}$
        Form the set of feasible processor assignments $F_p$ for task $i$ via eqns 3.3 and 3.4
        For every possible set $P_i$ such that $|P_i|=r_i$ and $P_i \subset F_p$ do
           For $a=1$ to $r_i$ do
              $q = a^{th}$ element of set $P_i$
              Find $CT_{i,a}(T_1,T_2,..,T_q \cup i,..,T_M)$ via equations 3-1, 3-2, and algorithm 3.1
           end do

           If ( $\max_{1 \le a \le r_i} CT_{i,a}(T_1,T_2,...,T_q \cup i,..,T_M)$+cost-to-go $\le$ UB then
              n=n+1
              f(n)= $\max\{\max_{1 \le a \le r_i} CT_{i,a}(T_1,T_2,...,T_q \cup i,...,T_M)$+cost-to-go, $(\sum_{n=1}^{N} s_n)/\sum_{m=1}^{M}(\mu_m)\}$
              $\tau_n=0$
              $\sigma_n = q$
           end if
        end do
        $Z_0 = Z - \{i\} \cup \alpha_i$
Step 3: m = arg $\min_{1 \le i \le n, i \notin C} f(i)$
        If $\tau_m = 0$ then
            For $q=1$ to $M$ do
                $T_q^m = T_q \cup \{\sigma_m\}$
            end do
        else
            For $i=1$ to $M$ do
                $T_q^m = T_q^{\tau_m}$
            end do
        end if
        $Z_m := Z_{\tau_m}$
        $C = C \cup \{m\}$
Step 4: If $Z_m = \phi$ output the result
        Select a task $i$ from $Z_m$ such that no processors of task $i$ appear in $Z_m$
        $Z_m = Z_m - \{i\} \cup \alpha_i$
        cost-to-go:= $l_i - \frac{s_i}{\mu_f}$
        Form the set of feasible processor assignments $F_p$ for task i via eqns 3.3 and 3.4
        For every possible set $P_i$ such that $|P_i| = r_i$ and $P_i \subset F_p$ do
            For $a = 1$ to $r_i$ do
                $q = a^{th}$ element of set $P_i$
                Find $CT_{i,a}(T_1^m, T_2^m, ..., T_q^m \cup i, ..., T_M^m)$ via eqn. 3-5, 3-6, and algorithm 3.2
            end do
            If( $\max_{1 \le a \le r_i} \{ CT_{i,a}(T_1^m, T_2^m, ..., T_q^m \cup i, ..., T_M^m)\}$ + cost-to-go $\le$ UB then
            n=n+1
            f(n)= max{ $\max_{1 \le a \le r_i}$ { $CT_{i,a}(T_1^m, T_2^m, ..., T_q^m \cup i, ..., T_M^m)$} + cost-to-go, f($\tau_m$)}
            $\tau_n = m$
            $T_q^m = T_q^m \cup \{ i \}$
        end if
    end do
    If $Z_m = \phi$ then C=C-{m}
    Go to step 3
```

Illustrative Example:

Consider a simple example where task and processor graphs as shown in figure 3-3. We use the heuristic algorithm to find an upper bound, UB=15.5, on the completion time, then start the search process. The search steps can be constructed as figure 3-4. If we assign a task to a processor where the HEF is greater than UB, the 5-tuples of that node will not be retained. Each time a leaf node with the lowest cost is selected for expansion. The number of processors assigned to a task at a node is equal to the replication number of that task. The cost at a node is taken from the maximum HEF among the costs of the same task at different processors and its parent task. The complete path from root node to the goal node corresponds to a mapping.

Although the A* algorithm provides us an optimal solution, the entire decision tree is usually very large. For a problem where a large number of nodes must be generated before an optimal solution can be determined, the A* algorithm can not be used to solve the problem due to the limitations of memory size and CPU time. Experience shows that in many problems A* algorithm spends a large amount of time discriminating among paths whose costs do not vary significantly from each other. In such cases the admissibility property becomes a curse rather than a virtue. It forces A* to spend a



TASK NUMBER	SECURITY MATRIX
1	1 1 1
2	0 1 1
3	1 1 1
4	1 1 1

FIGURE 3.3: AN ILLUSTRATIVE EXAMPLE FOR A* ALGORITHM

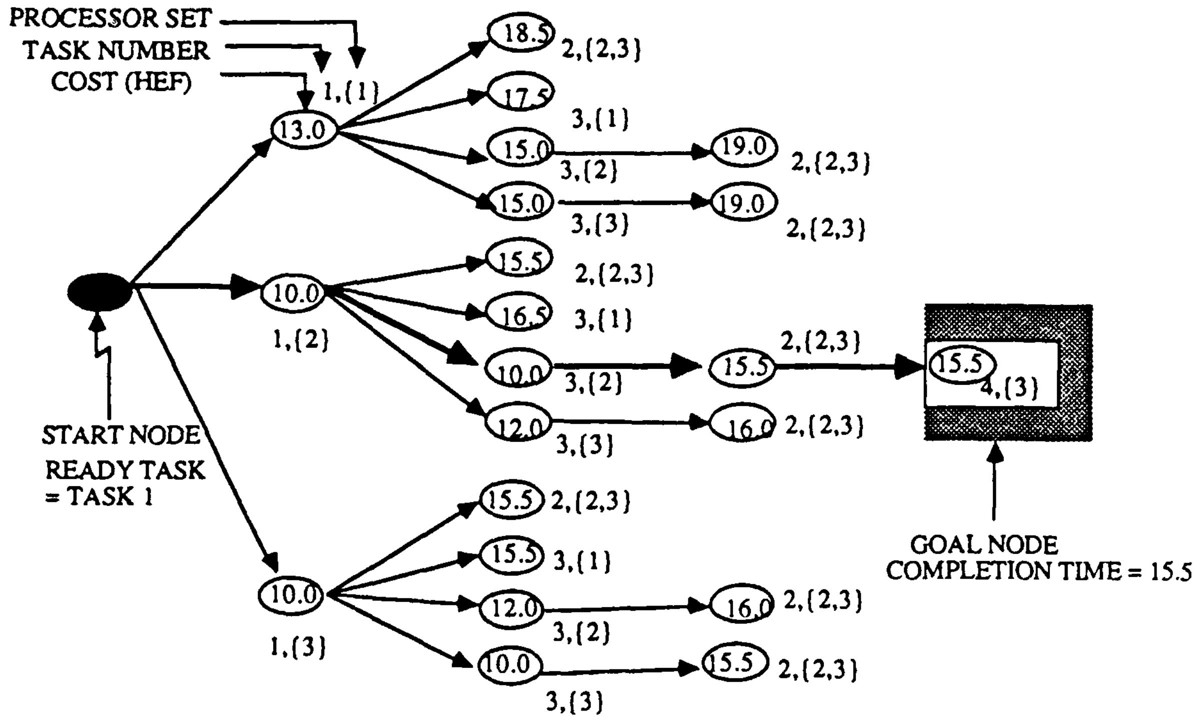


FIGURE 3.4: STATE SPACE SEARCH FOR A* ALGORITHM

disproportionately long time in selecting the best among roughly equal candidates and prevents A* trom completing the search with a suboptimal but otherwise acceptable solution. The A_ε^* algorithm is similar to the A* algorithm and is developed to compensate this drawback. In A_ε^* algorithm, ε is a nonnegative value which determines the deviation of the final solution to the optimal solution. In general, larger the value of ε, the less memory and CPU time is required to get the final solution. This value is used to control the speed of getting a solution from the decision tree. The main difference between the A* and the A_ε^* algorithms is the determination of node expansion order. In A* algorithm, the leaf node for expansion must be a node with a minimum HEF among the leaf nodes. In A_ε^* algorithm, the leaf node for expansion is a node with its HEF within $(1+\varepsilon)HEF_{\min}$ where $HEF_{\min}$ represents the minimum HEF value among the leaf nodes. In fact, the A* algorithm is a special case when ε=0.

Algorithm 3.7: A_ε^* *Algorithm.* Given a directed task graph $G_t = (V_t, E_t)$ with parameters (s_i, m_i, r_i, v_{ij}) and the processor graph $G_s = (V_s, E_s)$ with parameters (μ_p, R_p, c_{pq}), the A_ε^* algorithm computes the ε-optimal mapping $(T_1, T_2, ..., T_q, ..., T_M)$, where T_q is the ordered set of tasks allocated to processor q.

```
Step 1: Input ε
Step 2: Same as step 2 of algorithm 3.5
Step 3: $m = \arg \min_{1 \le i \le n, i \notin C} f(i)$
        mincost= $\min_{1 \le i \le n, i \notin C} f(i)$
Step 3.1: If we can find node m such that $f_m \le (1+\varepsilon) \times mincost$ then
            go to step 3.2
        else
            go to step 3
        end if
```

```
Step 3.2: If $\tau_m = 0$ then
        For $q=1$ to $M$ do
            $T_q^m = T_q \cup \{\sigma_m\}$
        end do
    else
        For $i=1$ to $M$ do
            $T_q^m = T_q^{\tau_m}$
        end do
    end if
    $Z_m := Z_{\tau_m}$
    $C = C \cup \{m\}$
Step 4: Same as algorithm 3.5 except go to step 3.1 instead of going to step 3
```

3.5 SUMMARY

In this section, we described the key mapping equations with and without constraints and the four mapping algorithms. The mapping due to task queueing, precedence relationships, data dependency among tasks, and message collision problems are explicitly considered in the key mapping equation. The heuristic algorithm employs the critical path method to determine a priority list of execution order and then uses the one-step optimization method to find a solution of the mapping. The pair-wise exchange algorithm is used to improve the solution of the heuristic algorithm by exchanging its priority list on the order of execution. Since the heuristic and pair-wise exchange algorithms do not guarantee an optimal solution, we develop an optimal mapping (A*) algorithm by considering all possible combinations of task execution order and allocation. A node with a minimum HEF is selected for expansion until a complete path consisting of all tasks is found. The path corresponds to an optimal mapping. The performance of the A* algorithm is limited by the required memory and CPU time. In order to reduce the requirements of CPU time and memory for the

A* algorithm, A_ϵ^* algorithm is developed to find a solution whose cost does not exceed the optimal cost by more than a factor $1+\epsilon$.

SECTION 4 COMPUTATIONAL EXPERIMENTS

4.1 EXPERIMENT 1: HYPOTHETICAL EXAMPLES

The first example involves the mapping of independent tasks onto identical processors. The task graph is a vertex graph with isolated vertices and no edges. In order to apply the mapping algorithms of section 3, a *dummy start task*, *task 10*, and a dummy terminal task, task 11, are added to the graph, as shown in Fig. 4-1. The service demands of these two dummy tasks are zero. The links between the dummy start task and the independent parallel tasks are added to the task graph. Similarly, links between the dummy terminal task and the independent parallel tasks are added. The amount of data transmitted between the dummy nodes and the other nodes is zero. The processor graph is a 2-cube computer system with unit service rate and unit link capacity. The levels of tasks are the same as the service demand in this example except for the dummy start task. The priority list then is (10,1,2,3,4,5,6,7,8,9, 11). Thus, scheduling independent tasks using the heuristic algorithm is equivalent to the LPT rule. The completion time of the heuristic algorithm is 15, which corresponds to the worst case mapping performance of the LPT rule. In the four processor case, the worst relative approximation error, $\frac{CT_{\max} - CT^{*}_{\max}}{CT^{*}_{\max}}$=0.25 If the pair-wise algorithm is used in this example, the completion time is improved to 13 and if the optimal mapping (A*) algorithm is used, the completion time is 12. The performance of the three algorithms is illustrated via Gantt Charts in Fig. 4-2.

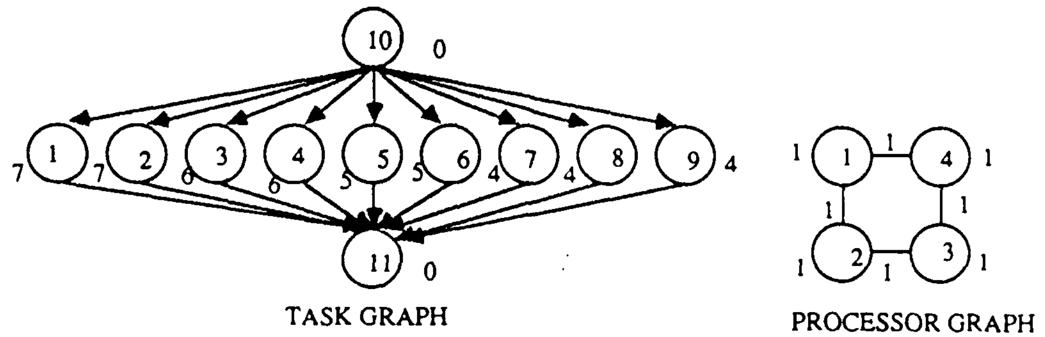
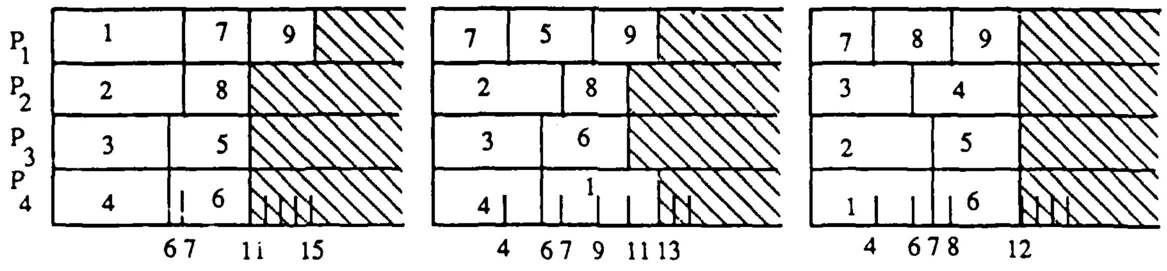


FIGURE 4-1: MODIFIED INDEPENDENT TASK GRAPH

The second example is related to the mapping of a tree-structured task graph with nonidentical service demands as shown in Fig. 4-3. A dummy start task, task 8, is added to the task graph. The links between task 8 and tasks 1, 2, 3, 4 are added into the task graph as in the previous example. The levels of the tasks are (4, 5, 3, 5, 2, 6, 1, 6) and the priority list constructed by the heuristic algorithm is (8, 6, 2, 4, 1, 3, 5, 7). The mapping provided by the heuristic algorithm is: T_1 = {8, 6, 1, 3, 5, 7} and T_2= {2, 4} with the concomitant completion time for the heuristic algorithm of 9 time units. If the pair-wise exchange algorithm is applied to solve this example the completion time is 8, which is an optimal mapping in this case. The optimal mapping is: T_1={8, 4, 1, 3, 5, 7} and T_2={2, 6}. Application of the heuristic algorithm to tree-structured tasks with arbitrary service demand for each task corresponds to critical path scheduling.



HEURISTIC ALGORITHM: COMPLETION TIME = 15   PAIR-WISE EXCHANGE ALGORITHM: COMPLETION TIME = 13   A* ALGORITHM: COMPLETION TIME = 12

FIGURE 4-2: GANTT CHART FOR THE EXAMPLE OF FIGURE 1

The third task graph is taken from Kasahara and Narita [1984] and is shown in Fig. 4-4. It is assumed that the amount of data transmitted among tasks is negligible. That is, precedence constraints among tasks are considered, but the communication among tasks is neglected. The level of the start task (in this case 9 time units) forms a lower bound for the completion time, i.e., no matter how many processors (with the service rate limited by the fastest service rate in the processor graph) are used to solve this problem, the completion time must be at least equal to the level of the start task. In this example, the completion time of the heuristic algorithm is 9 which is equal to the level of the start task. This means that the heuristic algorithm provides the optimal mapping in this case. If a completion time smaller than the level of the start task is desired, it can be accomplished in one of the two following ways: further partitioning of the tasks lying on the critical path into several parallel tasks such that the level of the start task is reduced or replacing the fastest processor in the architecture with an even faster service rate processor. However, decomposing a task into several subtasks may result in additional communication among subtasks.

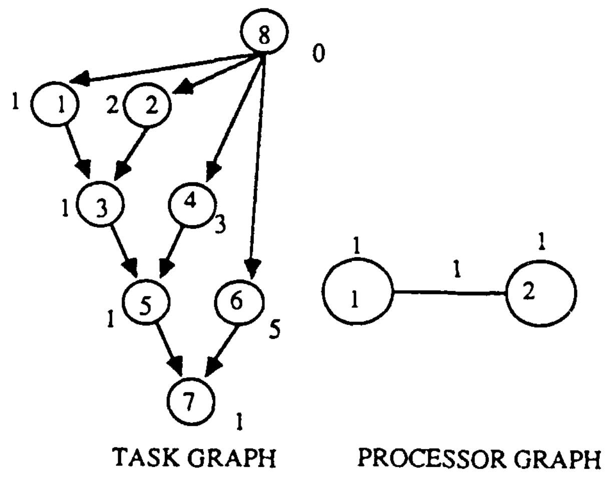


FIGURE 4-3: MODIFIED TREE-STRUCTURED TASK GRAPH

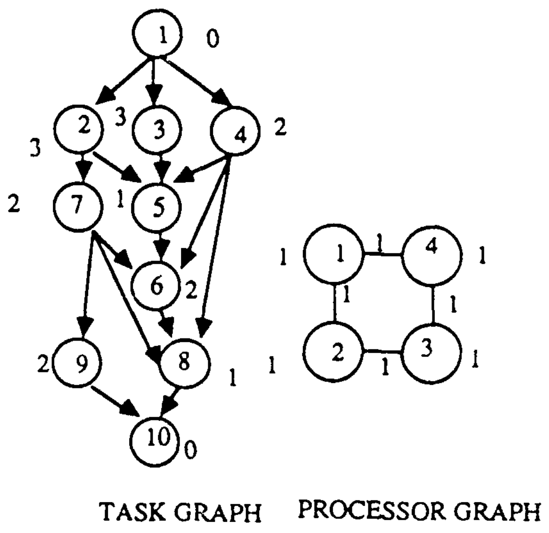


FIGURE 4-4: TASK GRAPH EXAMPLE FROM KASAHARA AND NARITA, 1984

The fourth example corresponds to a dynamic scene analysis algorithm and is taken from Agrawal [1986]. The task graph, shown in Fig. 4-5, consists of data dependencies and precedence relationships among tasks, a start task and a terminal task, and hence can be directly applied to our model. We used three algorithms to solve this problem. The completion time of the mapping with the heuristic algorithm is 23.60, the pair-wise exchange algorithm provides a mapping with a completion time of

23.33, and the A^*_ε algorithm provides a mapping with the completion time of 23.06, when ε=0.01. The A* algorithm, when applied in this example, did not terminate even after 4000 node expansions. Although we were unable to obtain the optimal solution for this problem, we knew that the solutions of the heuristic and pair-wise exchange algorithms are near the optimal solution, since the completion time of the mapping provided by the A^*_ε algorithm, 23.06, with $\varepsilon = 0.01$ is within 1% of the optimal completion time.

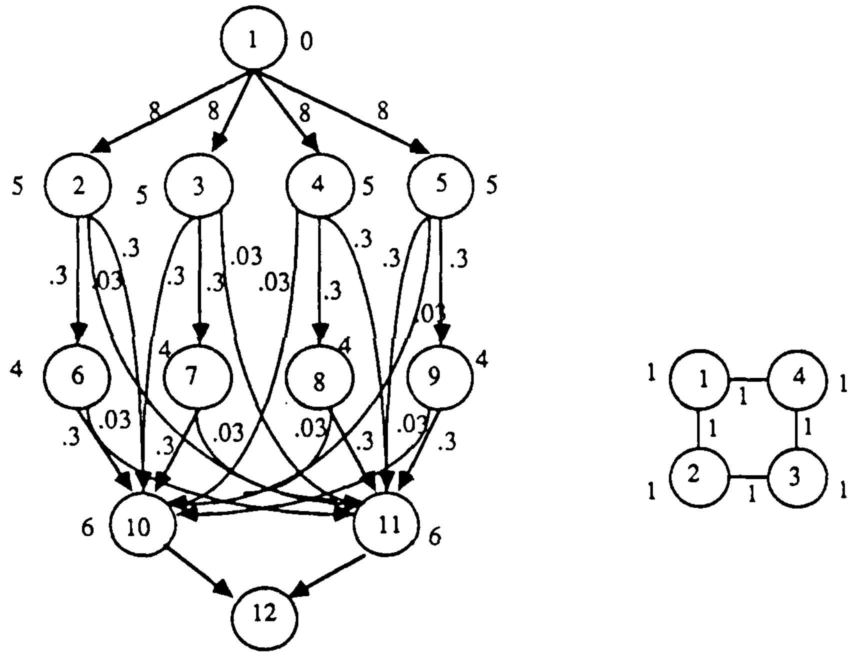
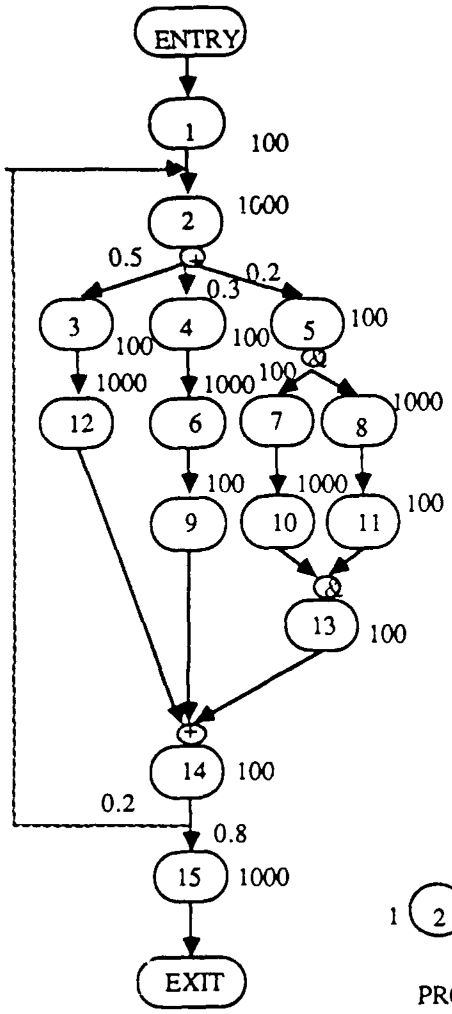
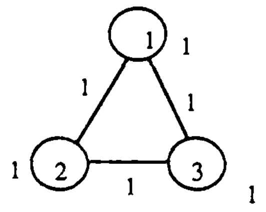


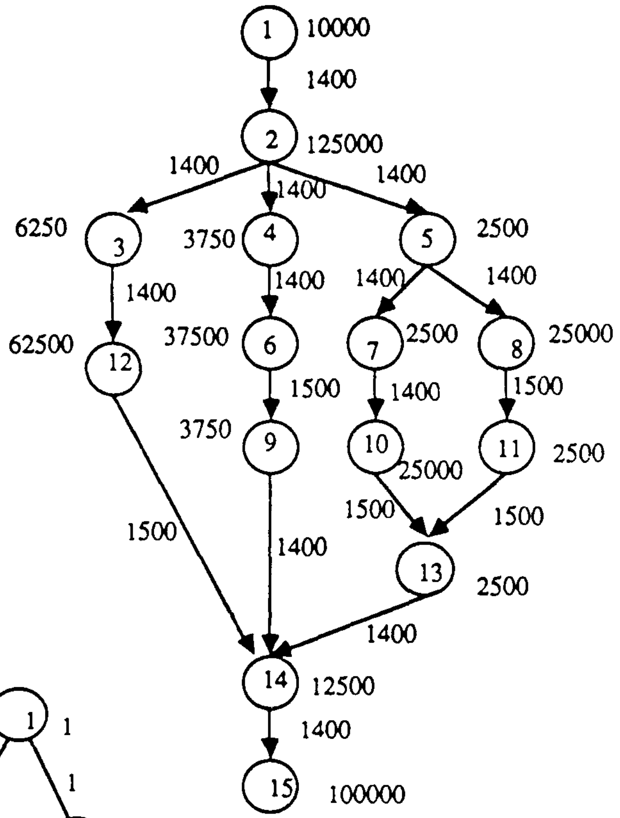
FIGURE 4-5: TASK GRAPH FROM ARGAWAL, 1986

The fifth example is taken from Chu and Lan [1987]. The control-flow graph, consisting of AND and OR nodes, is shown in Fig. 4-6. Every branch of the AND node should be executed. However, only one branch of the OR node is executed. The probability of executing each branch of the OR node is shown in Fig. 4-6. The control-flow graph can be converted into the task graph by computing the service demand of a task as the product of the number of visits and the service demand of the corresponding node in the control-flow graph. For example, the number of visits to





PROCESSOR GRAPH



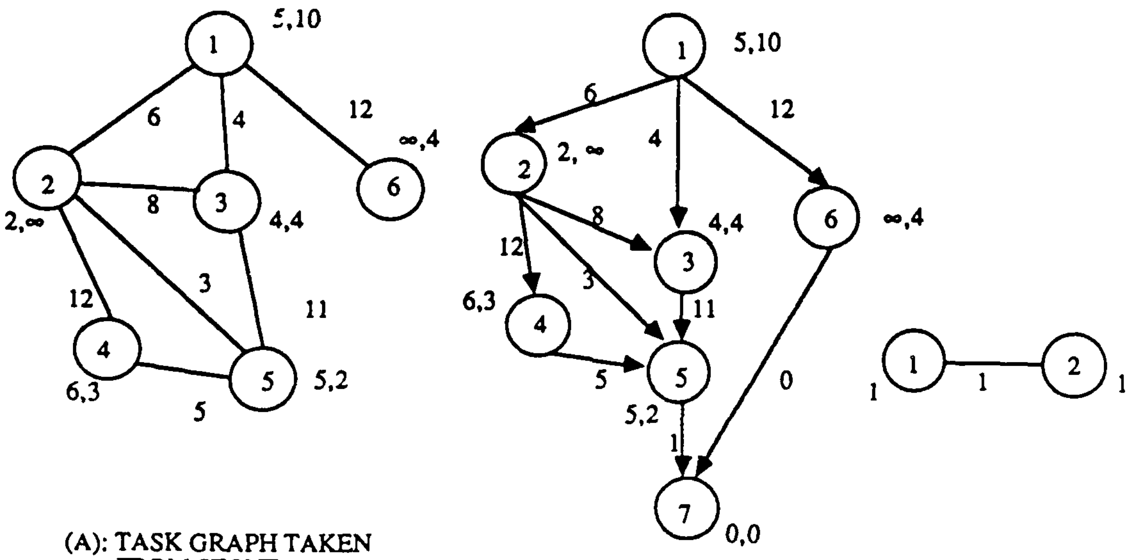
TASK GRAPH

FIGURE 4-6: CONTROL FLOW GRAPH EXAMPLE
FROM CHU AND LAN, 1987

FIGURE 4-7: MODIFIED TASK GRAPH
FOR FIGURE 4-6

node 2 in the control-flow graph is $1+0.2+0.2^2+0.2^3+....$ =1.25. The service demand of task 2 in the task graph is 1.25×1000=1250. Similarly, the service demand of task 3 in the task graph is equal to the number of visits, 1.25×0.5=0.625, times the service demand per visit, 100, which is 62.5. After 100 arrivals, the task graph is shown in Fig. 4-7. The link parameters are assumed to be 1400 or 1500. The completion time of the mapping provided by the heuristic algorithm is 316250 which, again, is equal to the level of the start task (or the minimum completion time) and is therefore optimal.

The sixth example shown in Fig. 4-8 (a) is similar to the task graph considered by Stone [1977]. We modify the task graph into a directed graph such that a directed edge starts from a lower numbered task to a higher numbered task as shown in Fig. 4-8 (b); note that task 7 is a dummy terminal task. The service demands and the amount of data transmitted among tasks are the same as those in Stone's example. Note that the computer system used by Stone is a heterogeneous system wherein the processing time of tasks does not correspond to a uniform system. We assume that the processors have identical service rates and the service demands of a task is different on different processors. The level of a task is determined from the level by taking the lowest service demand for each task. For example, the service demands of task 1 is (5,10), which means the service demand is 5 when task 1 is assigned to processor 1 and 10 when task 1 is assigned to processor 2. We take the service demand of task 1 as 5 when computing the levels of tasks in the task graph. Similarly, take service demand of task 2 as 2, 4 for task 3, 3 for task 4, and so on. The levels of tasks 1 to 7 are (13, 8, 6, 5, 2, 4 , 0) and the priority list of the heuristic algorithm is (1, 2, 3, 4 ,6 , 5, 7). The mapping provided by the heuristic algorithm is: T_1={1, 2, 3 , 4 , 5, 7} and T_2={6}. The completion time of the heuristic algorithm is 22. The computation time plus communication time is 26+12=38, which is the same as Stone's result.



(A): TASK GRAPH TAKEN FROM STONE

(B): MODIFIED TASK GRAPH FOR FIGURE 4-8(A)

PROCESSOR GRAPH

FIGURE 4-8: TASK GRAPH TAKEN FROM STONE, 1977 AND ITS MODIFIED TASK GRAPH

The final task graph, shown in Fig. 4-9, has not been addressed in the literature. Note that the processors have different service rates and the links have different capacities. The level of the start task is the sum of the service demands on the critical path, 210, divided by the fastest service rate, 2, which is 105 in this case. The levels of the other tasks can be derived from equation 3-7. The priority list of tasks is constructed according to nonincreasing levels first, and nonincreasing successors, if tasks have the same levels. The execution order then is (1, 2, 5, 4, 3, 6, 7, 8) and the corresponding levels are (105, 100, 100, 50, 50, 45, 45, 0). The highest level task, task 1, is assigned to one of the fastest processors, processor 2, with a completion time of 5 time units. The second task to be assigned is task 2 in this case. We assume that data communication takes place along the shortest delay paths. For example, suppose we send one unit of data from processor 2 to processor 4. If the routing path is (2,3,4) the communication time is $\frac{1}{c_{23}} + \frac{1}{c_{34}} = \frac{1}{2} + \frac{1}{2} = 1$. Instead, if the routing path is (2,1,4) the communication time will be $\frac{1}{c_{21}} + \frac{1}{c_{14}} = \frac{1}{1} + \frac{1}{4} = 1.25$. Hence, the shortest routing path should be (2,3,4). If task 2 is assigned to processor 1, the completion time will be the completion time of task 1 plus the communication delay from processor 2 to processor 1 plus the service time of task 2 at processor 1, and is equal to $5+\frac{10}{1}+\frac{110}{1}=125$. Similarly, if task 2 is assigned to processor 2, then the completion time of task 2 will be the completion time of task 1 plus the service time of task 2 at processor 2. That is, the completion time of task 2 at processor 2 is $5+\frac{110}{2}=60$. We can compute the completion time of task 2 at processors 3 and 4 from equations 3-1 and 3-2 in a similar manner. The best choice is processor 2. The next task will be task 5. Now if task 5 is assigned to processor 2 the completion time of task 5 will be the completion time of the last task that is assigned to processor 2 plus the service time of task 5 at processor 2 which is 115. In other words, if we can find a processor such that the completion time of task 5 on that processor is less than 115, we will assign to that processor. We

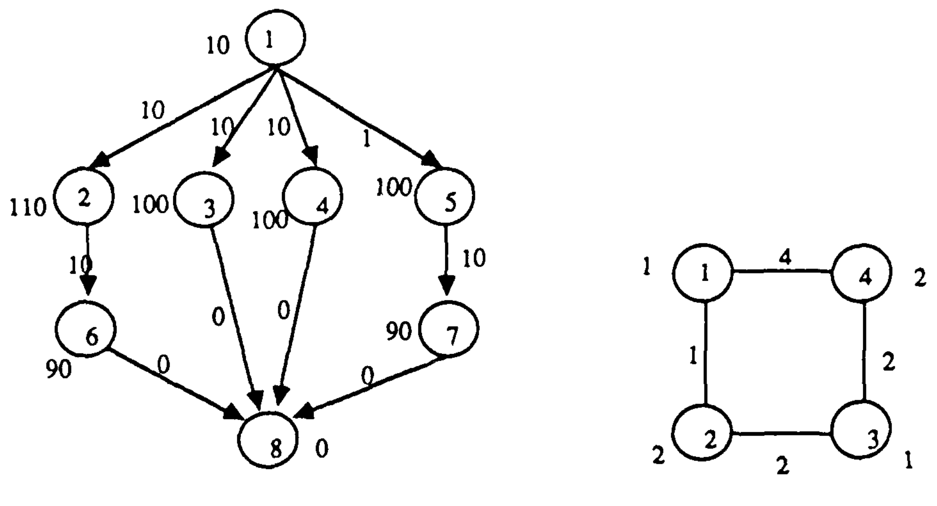


FIGURE 4-9: TASK AND PROCESSOR GRAPH FOR A HYPOTHETICAL EXAMPLE

find that assigning task 5 to processor 4 will give the earliest completion time of task 5, so processor 4 will be the best choice. The results oḟ the mapping algorithms are shown in table 4.1. The completion time of the mapping provided by the heuristic algorithm is 153.5. The pair-wise exchange algorithm provides a mapping with a completion time of 115, which is an improvement of over 30% for this example. The A^* (optimal) algorithm provides a mapping with a completion time of 110.5. Intuitively, the optimal mapping should allocate tasks 2 and 6 to processors 2 or 4, task 3 to processors 1 or 3, task 4 to processor 3, and tasks 5 and 7 to processor 2. This assignment will balance the processing time at different processors after task 1 has been completed. However, the heuristic or pair-wise exchange algorithms will not compute this mapping.

execution order	heuristic algorithm			pair-wise exchange algorithm			A* algorithm		
	task #	allocation	completion time	task #	allocation	completion time	task #	allocation	completion time
1	1	2	5.0	1	2	5.0	1	4	5.0
2	2	2	60.0	2	2	60.0	4	1	107.5
3	5	4	61.0	5	4	61.0	5	2	61.0
4	4	2	110.0	6	2	105.0	2	4	60.0
5	3	3	110.5	7	4	106.0	7	2	106.0
6	6	4	115.0	4	3	110.5	6	4	105.0
7	7	1	153.5	3	1	115.0	3	3	110.5
8	8	1	153.5	8	1	115.0	8	4	110.5

TABLE 4-1: COMPUTATIONAL RESULTS FOR DIFFERENT MAPPING ALGORTHMS

4.2 EXPERIMENT 2: RANDOM GRAPHS

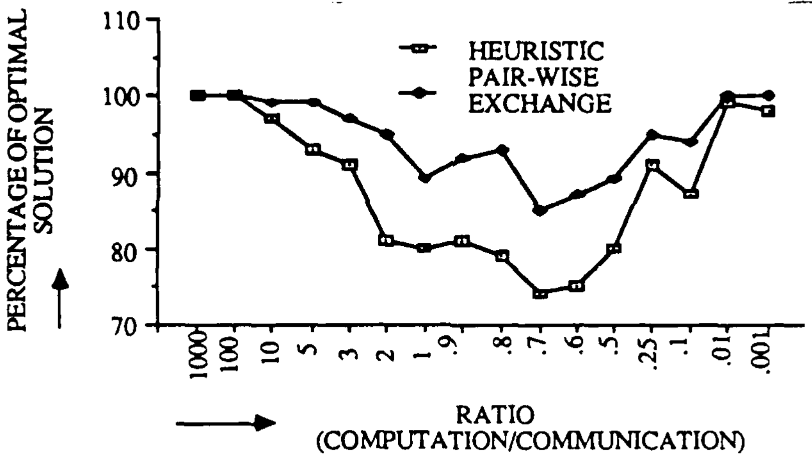
In order to understand the performance of the heuristic and pair-wise exchange algorithms, we performed an experiment on random task graphs. In this experiment, the computer architecture is a 2-cube system with unit service rate at each processor and unit link capacity. With this architecture, the service demand of a task and the amount of data transmitted among tasks can be treated as the computation time of that task and the communication time between tasks respectively. The task structure was generated as follows:

Algorithm 4.1: *Graph Generator Algorithm.* Given the ratio ρ of computation time / communication time, the graph generator algorithm generates random, directed, acyclic task graphs with number of tasks in the range [1,12].

```
Step 1: Generate a random integer number N in [1,12]
        Pick a start task j in [1,N]
        CLOSE={j}
Step 2: For i=1 to N do
            If $i \in$ CLOSE then
                Flip a coin to determine whether $i \in \alpha_j$
            end if
        end do
        If $|\alpha_j|=0$ then
            go to step 2
        end if
Step 3: Open := $\alpha_j$
        Pick a task k in OPEN
        CLOSE=CLOSE$\cup$k
        For i=1 to N do
            If $i \in$ CLOSE then
                Flip a coin to determine whether $i \in \alpha_k$
            end if
```

```
    end do
    If $|\alpha_k|=0$ then
        $\alpha_k=\{N+1\}$
    end if
Step 4: The node parameters are randomly selected in the range $[0,\rho]$
    The link parameters are randomly selected in the range $[0,1]$
```

We define the ratio ρ as computation time/communication time, and vary ρ in the range 0.001 to 1000. For each value of ρ, we generate one hundred random graphs as the random task graphs, and solve the mapping problem with the heuristic, pair-wise exchange, and A* algorithms. A notable result of the experiment is that the performance of the heuristic, pair-wise exchange, and A* optimal mapping algorithms are dependent on the value of ρ, i.e., the computation time/communication time. For large values of ρ (≥ 10) *the heuristic* algorithm tends to assign tasks to the first available processor. That is, the mapping tends to balance the work load on each processor. In this case, the percent test cases for which the heuristic algorithm provides an optimal solution is greater than 95%. As ρ decreases, the percent of test cases for which optimal mapping is provided by the heuristic algorithm decreases to 75% at the value of $\rho=1$ and then increases again. As ρ falls below 0.01, the heuristic algorithm provides optimal mapping for all random test cases. That is, the worst case performance of the heuristic algorithm occurs when $\rho=1$ (i.e., when the communication time is approximately equal to computation time). The pair-wise exchange algorithm provides optimal mapping in over 95% of the test cases except when ρ is near 1, where it provides optimal mapping in 85% of the test cases. The deviation of completion time of the heuristic algorithm to the optimal mapping may be very large at a small value of ρ (<0.01) because the communication time between a pair of tasks happens to be very small, and then becomes very large for the other pairs of tasks as compared to the computation time. Since the heuristic algorithm always assigns a task to a processor that has the earliest completion time of the assigned task, a local optimal mapping may result in large amounts of communication time when large amounts of data need to be transferred between two tasks in certain test cases. As for the pair-wise exchange algorithm, the deviation of completion time to optimal completion time is very small (below 2%) for almost all the test cases. The percentage of optimal mapping, average error versus different ratios of ρ are plotted in Fig. 4-10 . As for the optimal (A*) algorithm, the number of generated nodes and number of backtracks are small for small values of ρ (≤0.1) or large values of ρ (≥5). A^* algorithm generates maximum number of nodes and backtracks when ρ=1.



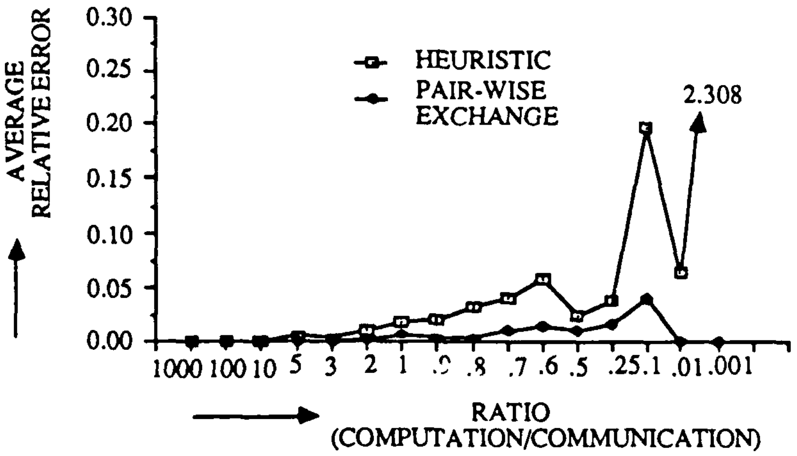


FIGURE 4.10: PERCENTAGE OF OPTIMAL MAPPING AND AVERAGE RELATIVE ERROR

4.3 EXPERIMENT 3: APPLICATION TO WEAPON TARGET ASSIGNMENT PROBLEM

This experiment involves the weapon target assignment and target sequencing (WTA/TS) problem arising in the boost phase battle management. As shown in Fig. 4-11, the weapon target assignment can be decomposed into a four level optimization problem by making use of the special features of the problem structure. The four levels are: (1) grouping targets into clusters based on their proximity to one another and the distinct launch tubes to which each target belongs, (2) determining the optimal allocation of directed energy weapons (DEWs) to the target clusters defined by level 1, (3) prescribing the optimal assignment of DEWs to individual targets within a cluster, and (4) determining the optimal fire control sequence of each target for each DEW.

The first level (target cluster definition) problem is formulated as a cluster median problem, which partitions the targets (both current and future expected) into groups such that the total sum of distances (or equivalently, slew times) in a cluster to the cluster median problem is minimized. The cluster median problem is a 0-1 integer programming problem, which is solved using Lagrangian relaxation techniques. The second and third levels (weapon-cluster allocation problem and weapon-target assignment within a cluster) are formulated as mixed-integer linear programming problems with the objectives of minimizing the leakage, balancing the allocation load among the weapons, and minimizing allocations that require large slew (switch-over) times. The problems, again, are solved via Lagrangian relaxation techniques [Korn et al., 1986]. Finally, the fourth level (target sequencing problem) is formulated as one of minimizing the weighted tardiness (i.e., value of leaked boosters) with sequence dependent setup (i.e., slew) times. The problem is solved via an approximate polynomially complete algorithm due to Ullman [1975] and Sahni [1976].

There are at least three salient features of the multi-level WTA/TS algorithm that are worth noting. First, the algorithm contains both serial and parallel subalgorithms.

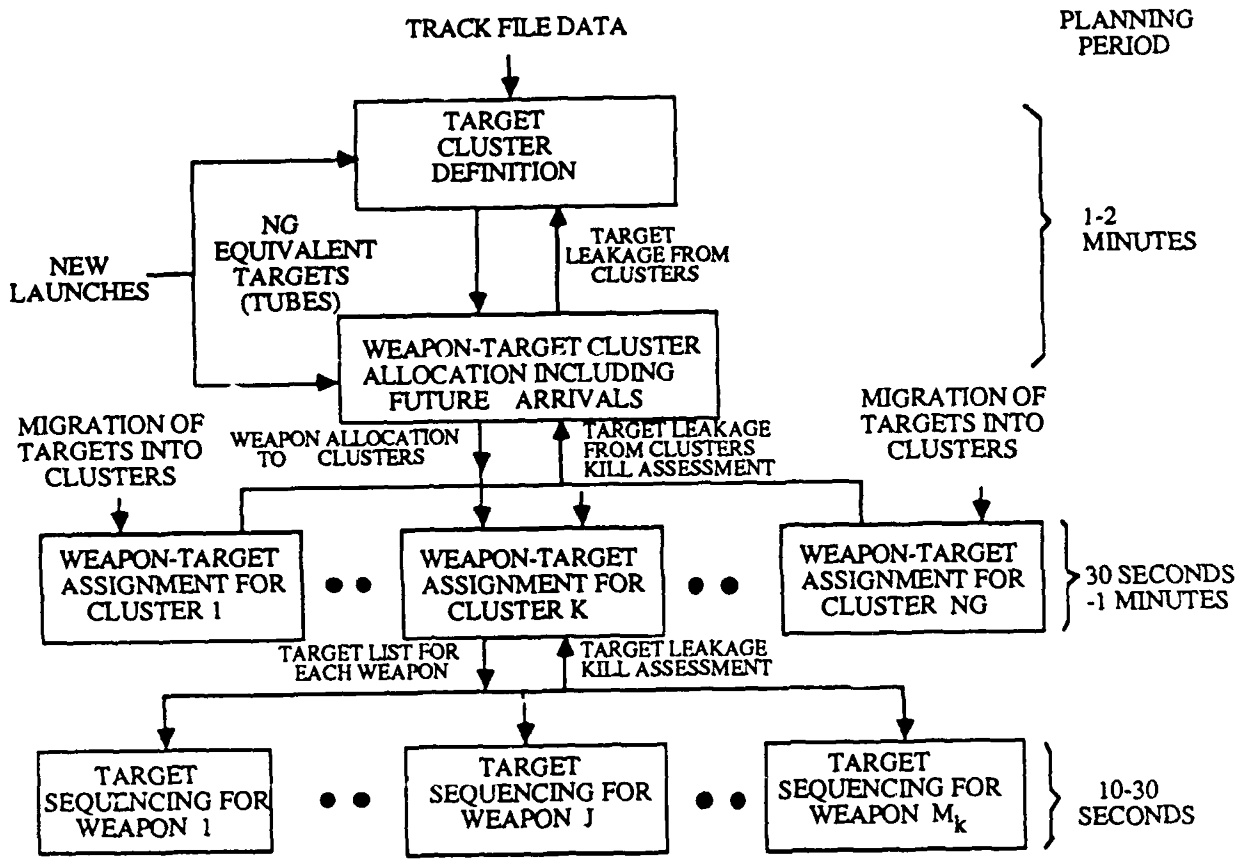

TRACK FILE DATA
PLANNING PERIOD
TARGET CLUSTER DEFINITION
1-2 MINUTES
NEW LAUNCHES
NG EQUIVALENT TARGETS (TUBES)
TARGET LEAKAGE FROM CLUSTERS
WEAPON-TARGET CLUSTER ALLOCATION INCLUDING FUTURE ARRIVALS
MIGRATION OF TARGETS INTO CLUSTERS
WEAPON ALLOCATION TO CLUSTERS
TARGET LEAKAGE FROM CLUSTERS KILL ASSESSMENT
MIGRATION OF TARGETS INTO CLUSTERS
WEAPON-TARGET ASSIGNMENT FOR CLUSTER 1
WEAPON-TARGET ASSIGNMENT FOR CLUSTER K
WEAPON-TARGET ASSIGNMENT FOR CLUSTER NG
30 SECONDS -1 MINUTES
TARGET LIST FOR EACH WEAPON
TARGET LEAKAGE KILL ASSESSMENT
TARGET SEQUENCING FOR WEAPON 1
TARGET SEQUENCING FOR WEAPON J
TARGET SEQUENCING FOR WEAPON M_k
10-30 SECONDS

FIGURE 4-11: OVERALL WEAPON TARGET ASSIGNMENT PROBLEM

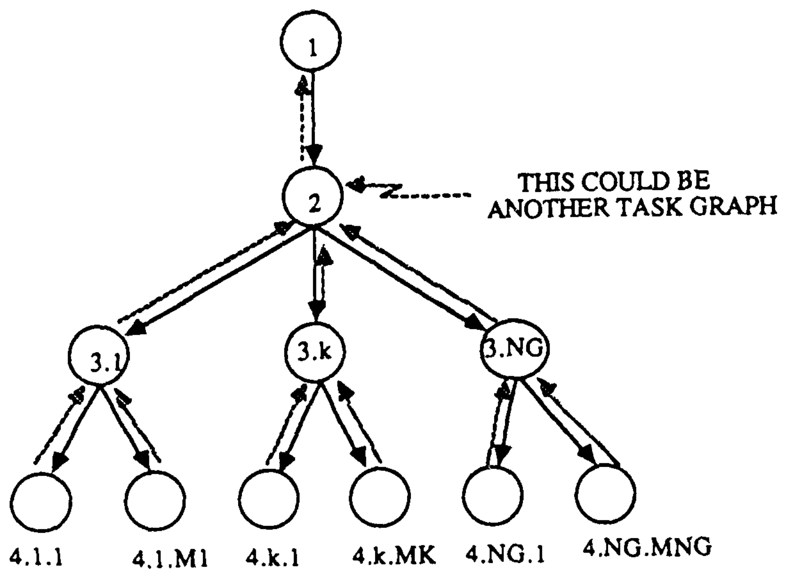


FIGURE 4-12: SUBALGORITHM LEVEL FOR WTA/TS PROBLEM

For example, when viewed as a whole, the cluster definition and the weapon-target allocation within a cluster and the target sequencing problems (i.e., levels 3 and 4) are parallelizable. In addition, portions of the Lagrangian relaxation approach used to solve the level 1 and level 2 problems are parallelizable. The key question then is: what constitutes a task of the WTA/TS algorithm, i.e., is it a subalgorithm or part of the subalgorithm? It is our contention that tasks should be defined as (appropriate) portions of subalgorithms to extract maximum parallelism. Second, the lower level subalgorithms have to periodically communicate with the higher level algorithms. Third, there exists time scale separation among the various levels, in the sense that higher level optimization subalgorithms are executed less frequently than the lower level optimization subalgorithms. For example, the target cluster definition and weapon-cluster allocation subalgorithms (i.e., levels 1 and 2) are executed every 1-2 minutes or whenever a new launch occurs. The level 3 (i.e., the weapon-target allocation within a cluster) subalgorithm is executed every 10-30 seconds to one minute, while level 4 target sequencing subalgorithm is executed every 10-30 seconds. Consequently, lower level subalgorithms impose more frequent workload on the fault-tolerant computer architecture than higher level ones.

Fig. 4-12 shows a task graph for the WTA/TS algorithm wherein each node represents a subalgorithm. However each subalgorithm can be a task graph in itself, if parallelism is exploited at a lower level of granularity. Once the level of granularity is selected, each task (or node) of the task graph is characterized by the number of instructions of each type of operation to be executed, and each link denotes the amount of data (in bits) to be transferred between the tasks. We use the following notations in the sequel.

term	meaning
NT	number of targets
NW	number of weapons
NG	number of clusters
NT_k	number of targets in each cluster
NW_k	number of weapons assigned to cluster k
NT_{kw}	number of targets assigned to weapon w in cluster k,
	$k=1, 2, ..., NG$; $w=1, 2,..., NW_k$
ND	number of dual iterations

TABLE 4-2: DEFINITION OF WTA VARIABLES

The service demands of tasks in the task graph can be approximated as follows:

task 1: $ND*NT+4*NT$ (clustering algorithm)

task 2: $ND*NW*(2*NG + 3) + (NG*NW)^2$ (weapon target clustering allocation algorithm)

task 3.k: $ND*NT_k*NW_k + (NT_k*NW_k)^2$ $k=1, 2, .., NG$ (weapon-target assignment within a cluster)

task 4.k.w: $(NT_{kw})^{order}$ $k=1, 2, ..., NG$; $w=1, 2,..., M_k$; order is the computational complexity of the fire control sequencing algorithm

The memory requirement of each task in the task graph is as follows:

task 1 : $3*NT^2 + 2*NT$

task 2 : $5*NT*NW$

task 3.k : $4*NT_k*NW_k$ $k=1,2,..NG$

task 4.k.w: $2^{NT_{kw}}$ for optimal solution

$4*NT_{kw}$ for weighted shortest processing time rule

Approximate link parameters: volume of data transfers between tasks i and j, v_{ij} (measured in number of words) are:

edge <1,2> : $NG+4*NT+4*NT*NW$

edge <2,3.k> : $4*NT_k*NW_k+2*NT_k+NW_k$; $k=1, 2, ..., NG$

edge <3.k,4.k.w>: $4*NT_k$, $k=1, 2, ..., NG$; $w=1, 2, ..., NW_k$

We assume that the weapon target assignment algorithm parameters are as follows:

(1) number of weapons in each cluster is uniformly distributed in [1,5]

(2) number of targets for each weapon is uniformly distributed in [1,50]

(3) number of dual iterations, $ND=100$

The number of clusters and order of fire control computational complexity are variable parameters. The processor graph is a 3-cube computer system with service rate and link capacity taken from the real N-cube computer, i.e., the service rate at each processor is 0.5 Mflops and the link capacity of each link is 10 Mbits/sec. In order to understand the dominant factors of the weapon target assignment, we made 35 runs with the number of clusters in the range [1,7] and the complexity of fire control sequencing algorithm as a function of the number of targets, NT_{kw}^{α}, with α in [1,5]. The test results are shown in table 4.3. It is clear from the results that the completion time increases considerably as α changes from 4 to 5, which means that the maximum admissible computation order of the fire control computational complexity should be less than or equal to 4. Otherwise it will take several minutes to process a subalgorithm and will not meet the real time requirements of the boost-phase WTA/TS algorithm.

4.4 EXPERIMENT 4: APPLICATION TO MULTITARGET TRACKING

Ballistic Missile Defense and Airborne Surveillance require identification and tracking of several hundred targets in real time. Multitarget tracking algorithms designed for these problems demand large computational resources which generally can't be fulfi'led with conventional von Neumann types of processors. However, due to the simultaneous tracking of several targets, the multitarget tracking algorithm will contain several computational tasks which may be run in parallel. With the percentage of optimal solution using the heuristic algorithmin in those thirty-five runs of randomly generatedweapon target assignment graph = 100 %

.5M FLOPS; 10M BITS/SEC ; 3 CUBE

order \ # of cluster	1	2	3	4	5	6	7
1	1.0000	1.0655	1.1477	1.1559	1.1830	1.4597	1.6944
2	1.0000	1.0917	1.1563	1.1669	1.1951	1.4774	1.7154
3	1.0000	1.6817	1.3221	1.4019	1.4462	1.7857	2.2555
4	1.0000	1.5700	1.7609	2.1274	2.2028	2.7994	3.0724
5	1.0000	1.3573	1.5676	1.9042	1.9440	2.4894	2.4726

SPEEDUP VERSUS # OF CLUSTERS
AND THE ORDER OF COMPLEXITY

ORDER	COMPLETION TIME (SECOND)						
	NUMBER OF CLUSTERS						
	1 T=31 W=1 TASK=5	2 T=96 W=4 TASK=9	3 T=208 W=9 TASK=15	4 T=241 W=10 TASK=17	5 T=278 W=12 TASK=20	6 T=355 W=16 TASK=25	7 T=456 W=19 TASK=29
1	0.0167	0.1351	0.8003	0.8147	0.8356	0.8797	0.9342
2	0.0188	0.1361	0.8035	0.8179	0.8388	0.8828	0.9374
3	0.0885	0.1626	0.9380	0.9524	0.9733	1.0173	1.0719
4	2.33885	2.4063	6.4516	6.4660	6.4869	6.5310	10.1984
5	78.2874	78.3053	232.5121	232.5265	232.5474	232.5915	459.1290

TABLE 4-3:COMPLETION TIME VERSUS NUMBER OF CLUSTERS
AND COMPLEXITY OF FIRE CONTROL ALGORITHM

development of multiprocessor architectures, adapting multitarget tracking algorithms to these new computer architectures poses several challenging problems including a good selection of computer architecture, task partitioning, and task allocations.

Implementation of the multitarget tracking algorithm requires the use of Kalman filters for updating target tracks and maintaining hypotheses corresponding to combinations of likely target tracks. Multitracker [Kurien, 1986] uses the mathematical framework of Hybrid State Estimation to formulate a solution methodology for the multitarget tracking problem. The general hybrid state model consists of continuous and discrete-valued states. Using measurements related to the hybrid state, it is possible to compute an optimal (minimum mean squared or maximum a posteriori) estimate of the hybrid state. 1

One of the current approaches toward this problem is termed track-oriented approach which provides a systematic methodology for constructing the optimal solution for multitarget tracking. However, for all practical scenarios which consist of several measurements in each scan, the computational requirements (both processing time and memory) will deplete the resources of any currently available computer. The reason for this problem is that the optimal algorithm postulates and retains all possible global hypotheses including the ones that are only remotely probable. In order to construct a practical algorithm, all such unlikely global hypotheses have to be eliminated. The key techniques incorporated in multitracker are (1) N-Scan Approximation, (2) Gating, (3) Classification, (4) Classification of Targets, and (5) Clustering. Details are discussed by Kurien [1986] .

Algorithm parameters are chosen on the basis of the anticipated scenario. For example, the number of confirmed targets accommodated by the algorithm should correspond to the maximum number of targets anticipated within the surveillance

1 We are grateful to Dr. Thomas Kurien for providing the data and multi-tracker algorithm of Fig. 4-13.

region; the number of tracks permitted for each target should take into account the clutter density, the proximity of other targets, and the probability of detection for targets. Similarly, the number targets and tracks per target permitted for Intermediate, Tentative, and Born targets should be based on target birth and death distributions and clutter distribution.

The major computational effort in multitracker is confined to three functional steps: (1) track predictions, (2) track updates, and (3) track pruning. The parameters used in multitracker are defined in table 4-4:

term	**meaning**
N_c	number of confirmed targets
N_i	number of intermediate targets
N_t	number of tentative targets
N_b	number of born targets
B_c	number of tracks per confirmed target
B_i	number of tracks per intermediate target
B_t	number of tracks per tentative target
N_r	number of returns per scan
NTC	number of targets per connected cluster
NGH	maximum number of global hypotheses
NS	number of subclusters
NTS	number of targets in each subclusters
NC	number of connected clusters
n	number of states modeled in the Kalman filter
m	number of measurements in each return

TABLE 4-4: DEFINITION OF MULTI-TRACKER VARIABLES

The computational requirements of each step are as follows:

prediction step : $1.5(n^3 + n^2)$ multiplications, and

$1.5(n^3 - n)$ additions

gating step : $N_r m + m(n^2 + 2n)$ multiplications, and

$2mN_r + m(n^2 + n - 1)$ additions

updating step : $1.5(n^3 + n^2)$ multiplications, and

$1.5(n^3 - n)$ additions

clustering step : $\frac{NTS\ (NS\ (NS-1))}{2}$ comparisons

global hypotheses: $min[(B_c + 1)^{NTC}, NGH]\ [\frac{NTC\,(NTC-1)NSCAN}{6}]$ comparisons

$min[(B_c + 1)^{NTC}, NGH]\ (NTC-1)$ multiplications

The operation count for pruning and promotion of targets, and for the creation of born targets is negligible.

Communication requirement for multitracker are listed as follows:

prediction step : $32n^2+48n+8$ bits

gating step : $16(n^2+N_r m)+32(n+m)+8$ bits

updating step : $16(n^2+N_g*m)+32(n+m)$ bits

clustering step : $16N_c*B_c*(NSCAN+1) + 4(N_c*B_c)$ bits

global hypotheses: $16NTC*B_c*(NSCAN+1)+36*NTC*B_c$ bits

A directed task graph summarizing the steps executed in one cycle of multitracker is shown in Fig. 4-13 . We assume the following requirements for arithmetic operations.

time for 32 bit multiplication	4 units
time for 32 bit addition	2.6 units
time for 16 bit comparison	1.3 units

Typical values for variables are:

$n=4$

$m=3$

$N_c = 100$

$B_c = 6$

$R = 1.5$

$N_i = 20$

$B_i = 3$

$N_t = 30$

$B_t = 3$

$N_b = N_r = 100$

$NTS = 2$

$NS = 50$

$NTC = 4$

$NC = 25$

$NGH = 100$

$NSCAN = 2$

$N_g = 1.5$

The associated computational requirements for various steps in Fig. 4-13 may then be evaluated as follows:

predict confirmed tracks	: 600* 714 time units
predict intermediate tracks	: 60* 714 time units
predict tentative tracks	: 90* 714 time units
predict born tracks	: 100* 714 time units
gate confirmed tracks	: 600*3196.2 time units
gate intermediate tracks	: 60* 3196.2 time units
gate tentative tracks	: 90* 3196.2 time units
gate born tracks	: 100*3196.2 time units
update confirmed tracks	: 900* 825 time units
update intermediate tracks	: 60* 825 time units

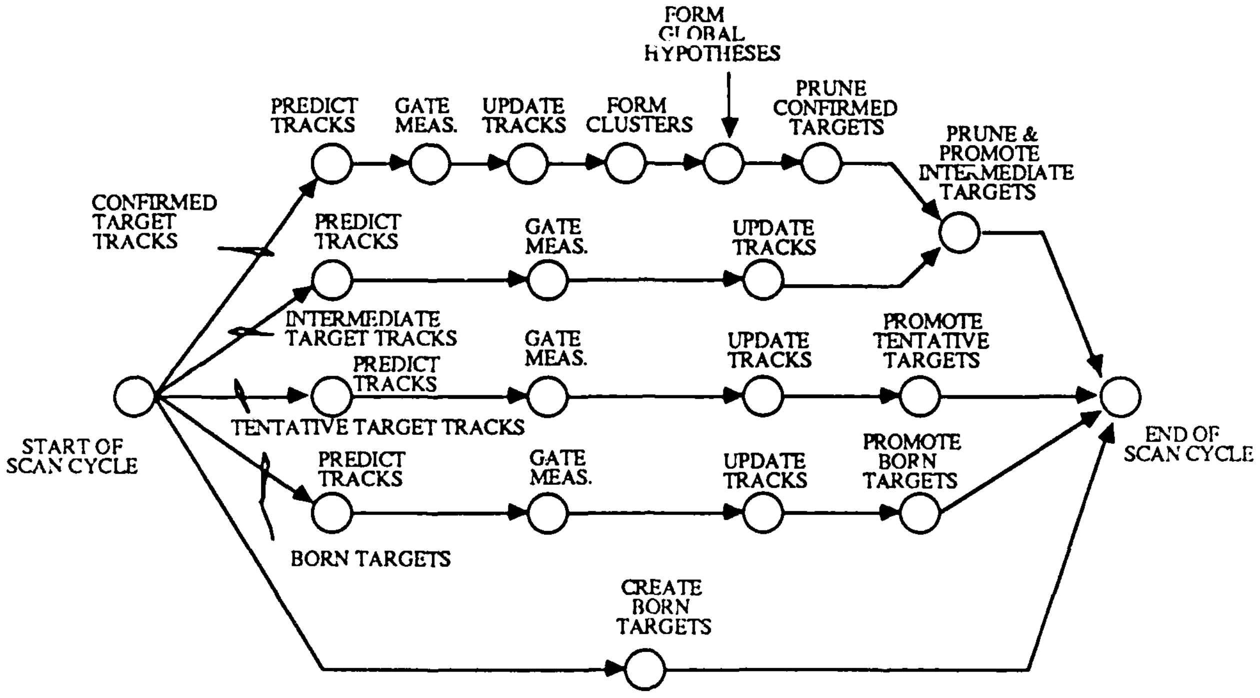


FIGURE 4-13: DIRECTED FLOW GRAPH OF OPERATIONS IN ONE SCAN OF TRACKING ALGORITHM

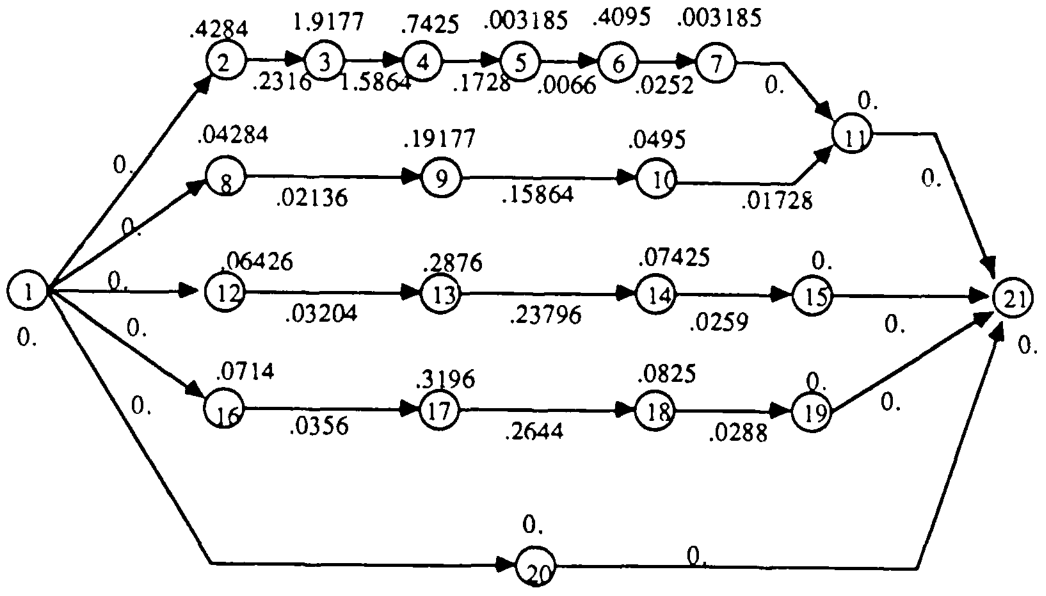


FIGURE 4-14: TASK GRAPH FOR AN ILLUSTRATIVE EXAMPLE

update tentative tracks	: 60* 825	time units
update born tracks	: 90* 825	time units
clustering	: 3183	time units
global hypotheses generation:	25*16380	time units

The associated data to be transferred are shown as follows:

prediction step	: 712	bits/track
gating step	: 5288	bits/track
updating step	: 576	bits/track
clustering step	: 13200	bits/track
global hypotheses step:	2016	bits/track

With these parameters listed as above, we can vary the granularity of a task and then using the algorithms described in section 3, we solve the associated mapping problem. If each node of Fig. 4-13 is defined as a task, then the task graph can be constructed as in Fig. 4.14. If we use 2-cube computer as the computational resource graph and use the heuristic algorithm to solve the mapping problem, we find that the completion time for one scan of the tracking algorithm is 3.5045 seconds and the corresponding speedup is 1.3378. Note that the completion time of the heuristic algorithm is equal to the level of the start task (or the maximum speedup is achieved), i.e., no matter how many processors are used to solve this problem, the maximum speed up is limited by the level of the start task. Nevertheless, since the nodes of the prediction, gating, and updating step are parallelizable (or can be decomposed into parallel subtasks), we can vary the granularity of a task size to decrease the start task level in order to get a better speed up. In order to understand the best task size for the multitracker, we partition the parallelizable tasks into subtasks thereby constructing a variable task graph. Heuristic and pair-wise exchange algorithms are employed to solve the corresponding mapping problem. At the beginning, we define 50 targets as a task

# of confirmed target per task	level	heuristic algorithm		pair-wise exchange algorithm	
		completion time	speedup	completion time	speedup
100	3.50	3.50	1.34	3.50	1.34
50	1.96	2.05	2.29	2.05	2.29
25	1.19	1.27	3.68	1.27	3.68
20	1.03	1.10	4.25	1.10	4.25
10	0.72	0.97	4.84	0.93	5.02
5	0.57	0.85	5.51	0.84	5.60
4	0.54	0.86	5.43	0.83	5.64
2	0.48	0.87	5.37	0.82	5.70
1	0.47	0.86	5.42	-----	-------

Table 4-5 (a): COMPLETION TIME AND SPEEDUP VERSUS NUMBER OF TARGETS FOR 3-CUBE MULTI-PROCESSOR SYSTEM

# of confirmed target per task	level	heuristic algorithm		pair-wise exchange algorithm	
		completion time	speedup	completion time	speedup
100	3.50	3.50	1.34	3.5045	1.34
50	1.96	2.05	2.29	2.0466	2.29
25	1.19	1.27	3.68	1.2744	3.68
20	1.03	1.49	3.14	1.4171	3.31
10	0.72	1.26	3.72	1.2249	3.83
5	0.57	1.24	3.79	1.2212	3.84
4	0.54	1.23	3.81	1.2155	3.86
2	0.48	1.26	3.73	1.2120	3.87
1	0.47	1.23	3.83	--------	-------

Table 4-5 (b): COMPLETION TIME AND SPEEDUP VERSUS NUMBER OF TARGETS FOR 2-CUBE MULTI-PROCESSOR SYSTEM in the confirmed path with all the other conditions fixed, we find the speed up increases by a factor of two. When 25 targets are defined as a task, the speed up is 3.68. If we continue partitioning the task size into smaller sizes (decrease the number of targets per task), we find that the speed up is limited to be 3.85. It is obvious that the speedup of a 4 processor computer architecture is at most 4. Therefore, we conjecture that the speed up may be limited by the architecture. Due to this reason, we change the architecture to a 3-cube computer architecture and make the same runs. The definition of 10 targets per task resulted in a speedup of 4.85. As the task size become smaller, the speed up curve levels off around 5.4. The detailed results are tabulated in table 4-4. Finally, we upgrade the architecture to a 16 processor computer system and repeat the partitioning experiment: the maximum speed up is 7.2.

The next experiment involves the selection of task size in every path to obtain a maximum speed up, i.e., the task partitioning of four different paths and global hypotheses simultaneously. If a 16 multiprocessor architecture is used, we find the maximum speed up for any possible decomposition of task size to be about 11.44 for the case when 5 targets make up a task in every path and 5 connected clusters form a task at global hypotheses step. Since the utilization of the processors is 0.7, we conclude that increasing the number of processors will not improve the speedup considerably. In other words, a multiprocessor architecture with 16 processors will be a good choice for solving the multitarget tracking problem.

In order to increase the speed up, tasks on the critical path (in this case, confirmed path) should be decomposed further. In this example, the decomposition of the other paths will not have a significant effect on the speedup. However, if we decompose the task to a very small size such that the critical path is no longer the confirmed path, or the highest level of this path is almost the same as that of any other path, then the decomposition of other paths may become important.

4.5 SUMMARY

In this section, we performed four experiments: (1) hypothetical example, (2) random graphs, (3) application to weapon target assignment problem, and (4) application to multitarget tracking. The first experiment involved the examples addressed in the literature. These examples can be applied to our mapping algorithms and the corresponding results showed that the heuristic mapping algorithm is a good methodology. In the second experiment, hundreds of random graphs with different ratios of computation time/communication time were generated for test examples. The test result showed that the heuristic algorithm provides optimal mapping when the ratio of computation time/communication time is large ($\geq$10) or small ($\leq$0.1). The worst test case occurred when the computation time is approximately equal to the communication time, wherein the heuristic algorithm provides optimal mapping in only 75% of the test cases. The pair-wise exchange algorithm provided optimal mapping in over 95% for almost all values of of computation time/communication time. In the third experiment, the mapping of a tree-structured weapon target assignment problem was considered. The service demands and amount of data transmitted among tasks were related to number of clusters, number of weapons, number of targets, and fire control complexity (α). The range of α in [1,5] together with the range of number of clusters in [1,7] were studied in this experiment. The results showed that unless the fire control complexity parameter, α, is less than or equal to four, it will not satisfy the real-time requirements of boost-phase WTA/TS. The percent of optimal mapping provided by the heuristic algorithm was 100% in these 35 runs. The last experiment involved the multi-tracker algorithm for multitarget tracking. In this experiment, the granularity of a task size and the selection of a computer architecture for the multi-target tracker problem were studied. The results showed that the critical path (confirmed path) dominates most of the computation time and, hence, the size of granularity for a task on the critical path and an appropriate computer architecture should be carefully selected in order to achieve the maximum speedup for the multi-tracker algorithm.

SECTION 5 CONCLUSIONS AND FUTURE RESEARCH WORK

5.1 CONCLUSIONS

In this report, we formulated the mapping problem as one of characterizing a BM/C^3 algorithm and the multi-processor architecture as graphs and of assigning and sequencing the nodes of the algorithm graph to the processor graph to minimize the completion time of the algorithm, subject to reliability, storage, and security constraints. The formulation explicitly considered precedence constraints, inter-task communication, security, storage, and reliability constraints. In addition, the processors are assumed to be uniform with different service rates and link capacities. A directed, acyclic task graph is used to denote an application algorithm whereas an undirected processor graph is used to denote multi-processor computer architecture.

The mapping problem is NP-complete, i.e., the memory and computational requirements of an optimal solution becomes intractable as the number of tasks and the number of processors becomes large. Therefore, all practical algorithms incorporate heuristics. In section 3, we derived four mapping algorithms viz., heuristic, pair-wise exchange, optimal (A*), and A_ε^* mapping algorithms. The key mapping equations that describe the evolution of completion time as a function of mapping are also derived in this section. These equations form the basis of all four mapping algorithms. The heuristic algorithm consists of two stages. The first stage employs the critical path method to determine task execution order while the second stage sequentially allocates the tasks from the ordered list to minimize the completion time. The pairwise exchange algorithm improves the performance of the heuristic algorithm by iteratively exchanging the order of elements in the priority list, without violating precedence constraints. Since the heuristic and pair-wise exchange algorithms do not guarantee an optimal solution, we developed an optimal (A*) algorithm. The A* algorithm searches for the optimal solution from a decision tree. This tree is created by considering all possible sequencing orders of execution and allocations. A path starting from the root node and ending at the goal node corresponds to the optimal mapping and the task sequencing at each processor. The completion time of the terminal task at the goal node is also the optimal completion time of the algorithm. In order to improve the search strategy of A* algorithm, the optimality criterion is relaxed in the development of the A_ε^* algorithm. The A_ε^* is ε-admissible, i.e., it always finds a solution with completion time that does not exceed the minimum completion time by more than (1+ε).

The algorithms are extensively tested via a set of four computational experiments in section 4. Experiment 1 was made up of hypothetical examples taken from the literature. Experiment 2 was concerned with the performance assessment of the heuristic and pair-wise exchange algorithms on random graphs. The test results showed that the performance depends on the ratio of computation time/communication time. The worst case performance of the heuristic and pair-wise exchange algorithms occurs when the computation time is approximately equal to the communication time. It was observed that the heuristic algorithm provides optimal mapping in 75% of the test cases and the pair-wise exchange algorithm provides optimal mapping in over 85% of the test cases in this case. When the ratio of computation time/ communication time is high or low (≥ 10 *or* ≤ 0.1), the heuristic and pair-wise exchange algorithms provide optimal mapping in almost 100% of the test cases. Experiment 3 was related to the application of the mapping algorithm to WTA/TS problem arising in the boost-phase battle management. This problem can be decomposed into a 4-level optimization problem. The task graph constructed according to the 4-level optimization problem results in a tree-structured graph. We varied the number of clusters in the range [1,7] and the computational order of fire-control sequencing algorithm in the range [1,5]. We found that the order of fire control sequencing algorithm should be less than or equal to 4. Otherwise, it will not meet the real-time requirements of the boost-phase BM/C^3. In experiment 4 we applied the mapping strategy to a multi-target tracking problem. It was observed that the completion time of the multi-target tracking algorithm is limited to the critical path of the task graph in some test cases. In order to maximize speed up, the granularity of a task should be carefully selected because inappropriate decomposition of a task may cause additional communication problems. Furthermore, we also discussed the issues of selecting a computer architecture for the multi-target tracking algorithm in this section.

5.2 FUTURE RESEARCH WORK

The future work involves: (1) quasi-static task allocation, (2) dynamic task allocation, and (3) task partitioning. The problem considered in this report was restricted to static task allocation problem, where the task graph $G_t(V_t\ , E_t)$ and the processor graph $G_p(V_p\ , E_p)$ are time-invariant. The quasi-static task allocation problem is concerned with the mapping of a sequence of task graphs $G_t(V_t\ , E_t\ , t_0\ , t_1 \dots t_{i-1}\ , t_i \dots t_n)$ onto a sequence of processor graphs $G_p(V_p\ , E_p\ , t_0\ , t_1 \dots t_{i-1}\ , t_i \dots t_n)$. The task graph may change due to the completion of existing tasks or arrival of new tasks. The processor graph may change due to the failures, and/or recovery of processors and/or communication links. However, in a given time interval $(t_{i-1}\ , t_i)$, i=1 ,..., n, the task graph $G_t(V_t\ , E_t)$ and the processor graph $G_p(V_p\ , E_p)$ are stationary. This problem can be solved by solving a sequence of static mapping problems in intervals $(t_0\ , t_1)\ , (t_1\ , t_2) \dots (t_{i-1}\ , t_i) \dots (t_{n-1}\ , t_n)$, while taking into account migration delays.

The dynamic task allocation problem involves the mapping of a task graph $G_t(V_t\ , E_t\ , t)$ onto a processor graph $G_p(V_p\ , E_p\ , t)$ where task and processor graphs are time varying. The time varying feature of the processor graph is associated with automatic fault-detection, isolation, and reconfiguration strategies. To solve this problem, we must consider the effects of current mapping decisions at time t on all future assignments and processor configurations. This multi-stage optimization problem can be approached via approximate dynamic programming.

Task partitioning is related to the granularity of tasks and domain decomposition. The partitioning of a task into several parallel subtasks may increase or decrease the speed up of an algorithm due to the parallelism and the communication among those subtasks. The best task partitioning should provide us a task graph that maximizes the speedup on a given computer architecture.

APPENDIX A

A.1 MAPPER USER INTERFACE

MAPPER is an interactive graphical environment to analyse task allocation in multiprocessor systems. It is a design tool which allows the user to evaluate alternative partitioning and mapping strategies onto various computer architectures in a systematic, intutive and interactive manner.

MAPPER is a user-friendly package. The user sketches the task graph (computational flow graph - CFG) and the processor graph (computational resource graph - CRG) using the mouse. Then, the user can configure the program, by selecting appropriate alogrithms and perform the mapping of CFG onto the CRG. The MAPPER displays the results in the form of a Gantt chart along with other performance measures (speed-up, completion time and processor utilization).

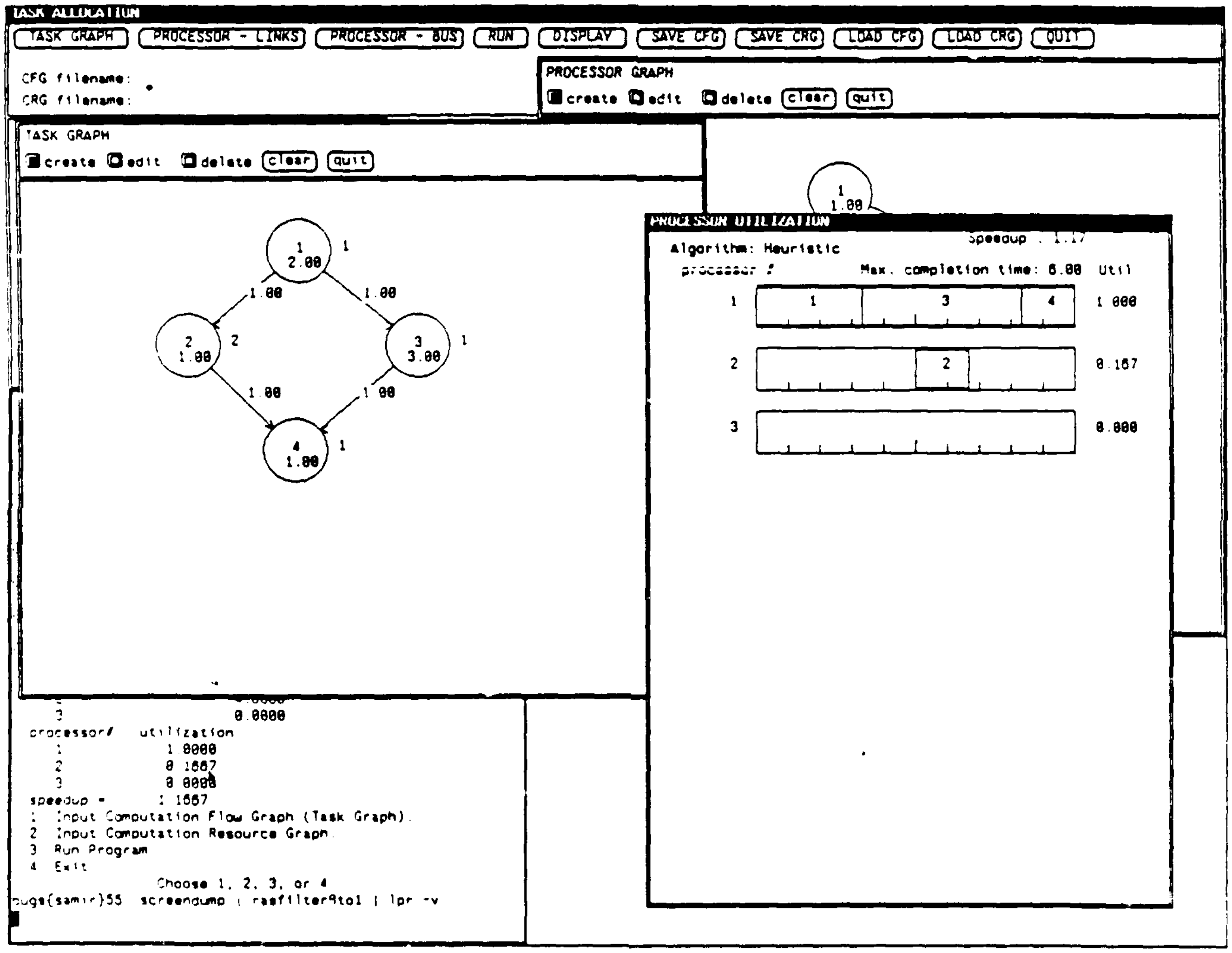
MAPPER is hosted on the SUN workstation and can be executed in Suntools environment. The user interface for the MAPPER conforms to any suntools application program (iconedit, mailtool etc.).

A.2 DESCRIPTION

MAPPER is a flexible design environment that allows the user to switch back and forth through different functions. Specifically the program has backtracking and editing capabilities,thereby allowing the user to fine-tune a particular problem rapidly without having to leave the environment.

MAPPER is like any suntool application. It is made up of six overlapping windows.

1. Main Command window.
2. Task graph window.
3. Processor graph window.
4. Algorithm selector window



```
                                        Speedup . 1.17
 Algorithm: Heuristic
   processor #           Max. completion time: 6.00  Util
          1   |    1    |      3      |  4  |   1 000
          2   |              |  2  |          |   0.167
          3   |                             |   0.000
```

```
   3                    0.0000
 processor#   utilization
   1              1.0000
   2              0 1667
   3              0 0000
 speedup =        1 1667
 1  Input Computation Flow Graph (Task Graph).
 2  Input Computation Resource Graph.
 3  Run Program
 4  Exit
                Choose 1, 2, 3, or 4
bugs{samir}55  screendump | rasfilter8to1 | lpr -v
```

FIGURE A-1: A SCREEN ENCOUNTERED IN A TYPICAL MAPPER WORK SESSION

5. C-shell.

6. Results window.

A typical screen encountered in a MAPPER session is shown fig. A-1. The figure shows the Task Graph window, the top part of the Processor graph window, part of the C-shell and the Results window in the foreground.

The MAPPER works as follows. After invoking the program the user clicks on the main selction panel to open the task graph window. The task graph can then be drawn on the screen using the mouse. The data-entry window pops up automatically whenever a node or a link data is required. The task graph window has editing features (add nodes, links; delete nodes, modify numerical data, destroy the whole graph). The processor graph window is functionally identical to the window described above except that the user has a choice of either opening this window in a bus-architecture or a link architecture mode.

The main selection has LOAD/SAVE features for both the graph windows. This feature is particularly useful to create a library of processor graphs. Standard architectures can be loaded when required instead of having to draw them each time the program is used.

The algorithm selector window configures the mapping algorithms to operate in the "with constarints mode" or the "without constraints mode". After the selection is made, the user can specify the algorithm to be used to perform the task allocation. MAPPER computes the task allocation and presents the results in the C-shell window.

The visual display can be invoked by clicking on the main selection panel. The final result is displayed as Gantt chart for each processor with the relevant performance measures.

The user can cycle through the stages described above interactively. This coupled with the editing features allows the user to compare, analyse and evaluate the mapping by incrementally modifying CRGs, CFGs or by using different mapping algorithms.

A.3 SOFTWARE DESCRIPTION

MAPPER is written as a two-level program.

1. Mathematical algorithms

2. Graphical interface.

The mathematical algorithms consist of Heuristic, pair-wise exchange, A-star and approximate A-star mapping algorithms and are implemented in FORTRAN. These algorithms can operate independently of the graphical interface with a keyboard-screen based I/O or input through ASCII datafiles. This algorithm program forms the basis of MAPPER.

The graphical interface is a Sunview based graphical shell written in C. The graphical program enables the user to access the mathematical algorithms in an interactive mode. Instead of inputting the data in numerical format, the user can draw the CRG and the CFG. The interface converts these drawings into an ASCII datafile, which is read by the FORTRAN program. Similarly the result file from the algorithm program is read by the graphics program and converted into graphical results. The graphical program also serves to make the whole package into an environment where the user can work interactively.

APPENDIX B

USER'S MANUAL:

MAPPER is a mouse-driven, window based suntool application. The reader of thi appendix is assumed to have working knowledge of any suntool application like Dbxtool or Iconedit etc.

Section B.1 describles the compilation procedure for the source code. This would be required only if the source code is modified. Section B.2 desribes theusage of the program including the definition and function of all the panel buttons.

B.1 COMPILATION

The source code can be compiled as follows,

```
cc -o mapper mapper.c -lsunwindow -lsuntool -lpixrect -lm
```

B.2 PROGRAM DESCRIPTION

Run the program from suntools by typing mapper.

Make sure that the algorithms program "alloc.out" is in your directory.

Associated with each screen is a panel displaying choice buttons. The user can activate a particular function by clicking on the corresponding choice button.

Choice Button functions: a) Main selection window:

1. TASK-GRAPH: Clicking on this button opens the task graph (CFG) window. The graph can be drawn on this canvas-window. The edit, delete and draw button acivate the respective modes of mouse operations. QUIT closes the window and CLEAR destroys the window and the data associated with it.

2. PROCESSOR-GRAPH LINKS: opens the processor graph window. This window is to be used to draw message passing architectures. The functions are identical to those of Task graph window.

3. PROCESSOR-GRAPH BUS: is a similar window used to draw bus-architectures.
4. RUN: generates the files required for the algorithm program and displays the algorithm selection panel.
5. DISPLAY: clicking on this opens the display window and displays the mapping results. This window should be opened after the mapping program alloc.out has terminated.
6. SAVE CRG/ save CFG: saves the CRG/CFG screens and the data. Make sure that you enter the CRG filename and CFG filname before invoking this function.
7. LOAD CRG/ LOAD CFG: performs the corresponding retrieval of the CRG and the CFG screens/data from the named files.
8. QUIT: destroys all the windows and exits to suntools. All data is lost if not saved prior to QUIT.

The main selection panel and the algorithm selection panel are shown in fig. B-1.

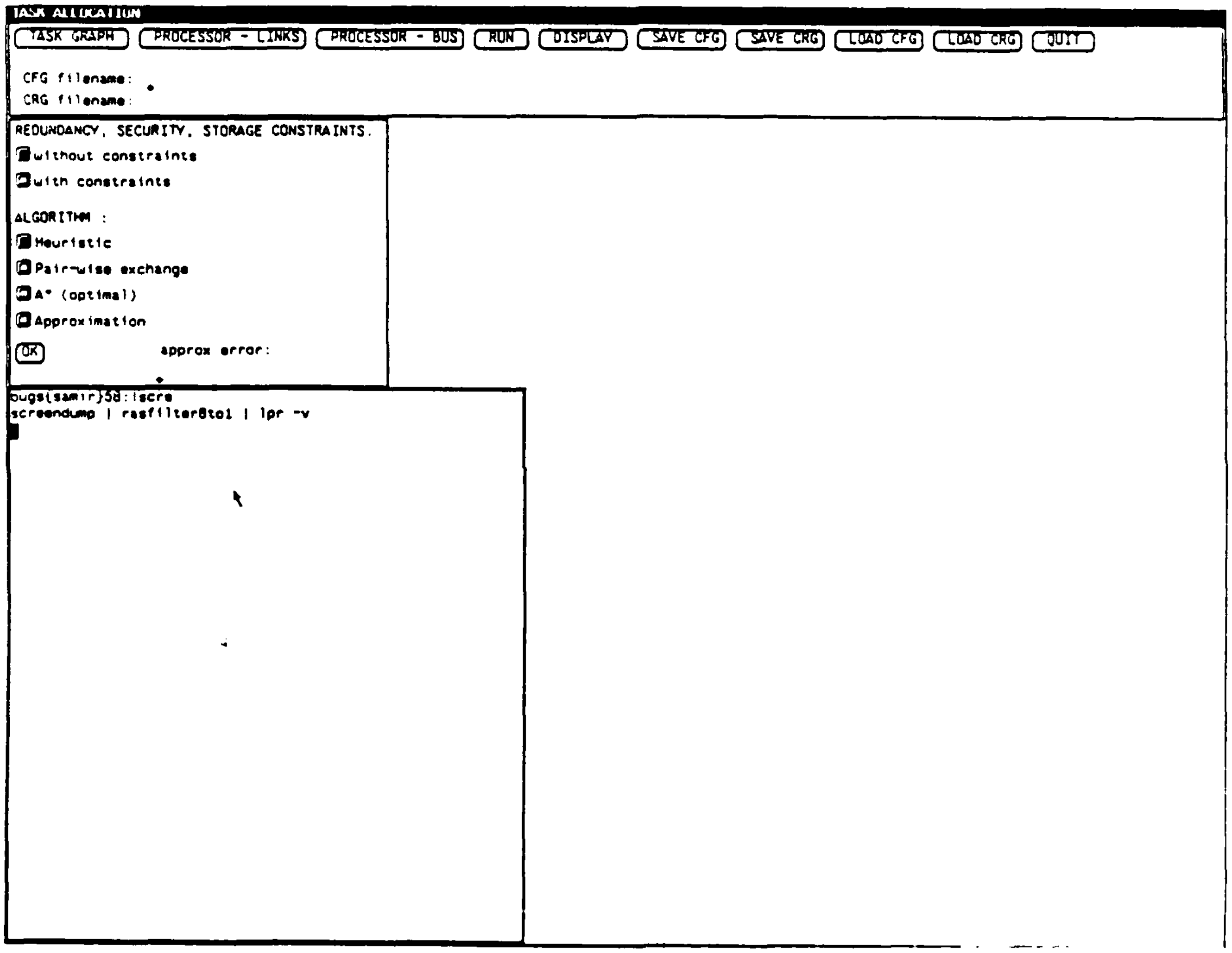
B.3 DRAWING GRAPHS:

The procedure to draw the task graph and the processor graph is similar.

On entering the particular window (task, processor) make sure that the draw button is highlighted that is you are in the draw/add mode.

Clicking the MIDDLE button of the mouse draws a node (circle) at the mouse-pointer. The data-entry window pops up. The user has to enter data and click on OK before proceeding.

After drawing more than two nodes a link can be drawn by clicking on the LEFT button at the two nodes to be linked. Enter the data as usual and close the data window by clicking OK.



CFG filename:
CRG filename:

REDUNDANCY, SECURITY, STORAGE CONSTRAINTS.
[✓ without constraints]
with constraints

ALGORITHM :
[✓ Heuristic]
Pair-wise exchange
A* (optimal)
Approximation

approx error:

```
bugs{samir}58:!scre
screendump | rasfilter8to1 | lpr -v
```

FIGURE B-1: MAIN SELECTION WINDOW, ALGORITHM SELECTION PANEL AND C-SHELL

EDIT mode is meant to modify data associated with either a node or a link. On entering this mode (highlight edit button by clicking on it), click the MIDDLE button at a node. The data entry window pops up. Modify required entries and close the pop-up window. Clicking the LEFT button at two nodes will make the data-entry window associated with that link to pop-up. Modify entry and close the pop-up window.

The data entry window for the task node is shown in fig B-2 and for the processor node in fig B-3.

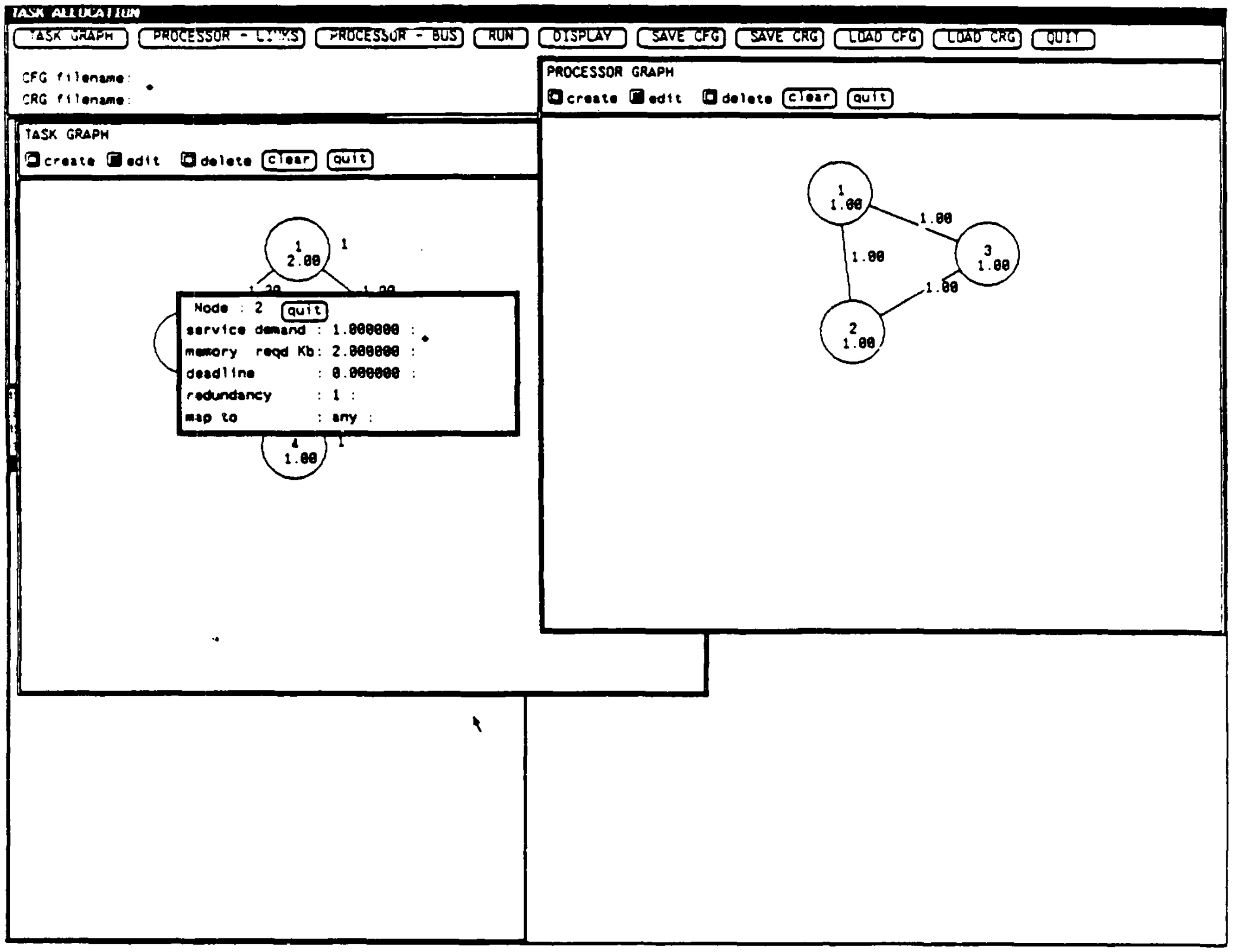
DELETE mode is used for modifying the graph by deleting nodes. After entering this mode (by clicking on delete), clicking the MIDDLE button at a node results in the deletion of that node and the links associated with it. The node indices are renumbered.

ALGORITHM SELECTION PANEL: This panel allows the user to select the mapping algorithm (with or *without constraints*)and finally execute it by clicking OK.

The C-shell window allows access to UNIX commands without having to leave the environment. The way to invoke other suntool applications while in MAPPER is to run the other application as a background process. For example to invoke the text editor type "texteditor &" in the C-shell window. The editor will be overlayed over the MAPPER window.

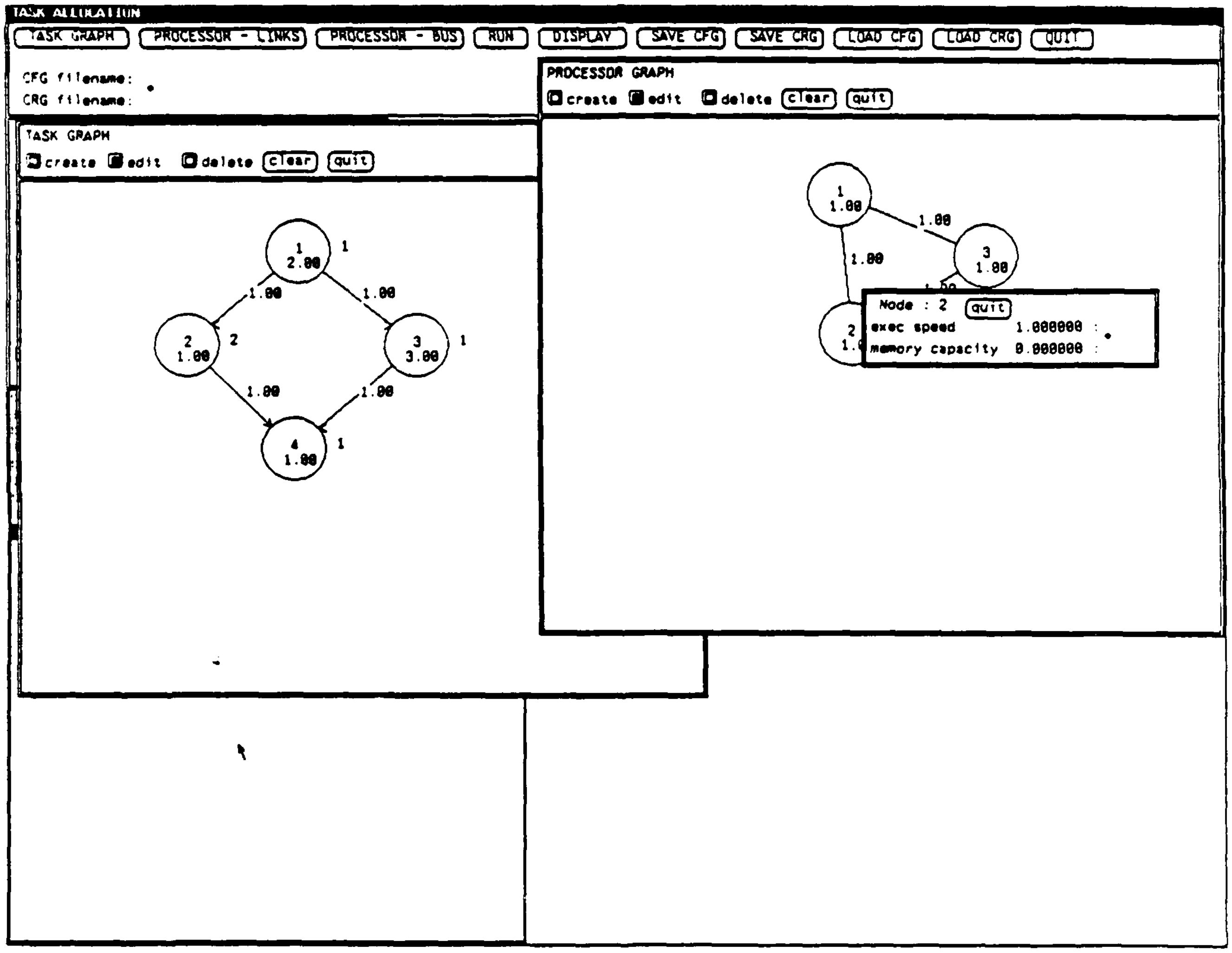
The screendump (hardcopy of the screen) can be made by entering the follwoing command in the C-shell. (note: make sure that the screen does not change during this process)

```
screendump | rasfilter8to1 | lpr -v
```



```
Node : 2 (quit)
service demand : 1.000000 :
memory  reqd Kb: 2.000000 :
deadline       : 0.000000 :
redundancy     : 1 :
map to         : any :
```

FIGURE B-2: TASK NODE ENTRY WINDOW



```
Node : 2   quit
exec speed        1.000000 :
memory capacity   0.000000 :
```

FIGURE B-3: PROCESSOR DATA ENTRY WINDOW

REFERENCES

Agrawal, D. P., Janakiram, V. K., and Pathak, G. C., "Evaluating the Performance of Multicomputer Configurations," *IEEE Computer*, Vol. 19, No. 5, May 1986, pp. 23-37.

Bokhari, S. H., "On the Mapping Problem," *IEEE Trans. Computer*, Vol. C-30, Mar. 1981, pp. 207-214.

Bokhari, S. H., "A Shortest Tree Algorithm for Optimal Assignments across Space and Time in a Distributed Processor System," *IEEE Trans. Software Eng.*, Vol. SE-7, No. 6, Nov. 1981, pp. 583-589.

Chu, W. W., Holloway, L. J., Lan, M. T., and Efe, Kemal, "Task Allocation in Distributed Data Processing," *IEEE Computer*, Nov. 1980, pp. 57-69.

Chu, W. W., "Optimal File Allocation in a Multiple Computing System," *IEEE Trans. Computers*, Vol. C-18, Oct. 1969, pp. 885-889.

Chu W. W.,and Lan, L. M-T, "Task Allocation and Precedence Relations for Distributed Real-time Systems," *IEEE Trans. on Computers*, Vol. C-36, No. 6, June 1987, pp. 667-679.

Coffeman, E., Garey, M., and Johnson, D., "An Application of Bin-packing to Multi-processor Scheduling," *SIAM Computing*, Vol. 7, No. 1, 1978, pp. 1-17.

Cook, S. A., "The Complexity of Theorem-proving Procedures," *Proceeding of the Third ACM Symposium on Theory of Computing*, 1971, pp. 151-158.

Ford, L. R., Jr., and Fu!kerson, D. R., "Flows in Networks," *Princeton University Press*, 1962.

Frenk J. G. B., and Rinnooy Kan, A. H. G., "The Asymptotic Optimality of the LPT Rule," *Math. Oper. Res.*, Vol. 12, No. 2, May 1987.

Garey, R. M. and Johnson, D. S., "Computers and Intractability," *W. H. Freeman Co.*, San Francisco, 1979.

Graham, "Bounds on Timing Anomalies," *SIAM J. on Applied Math.*, 17 (2), pp. 416-429, 1969.

Horowitz, E. and Sahni, S., "Fundamentals of Computer Algorithms," 1978.

Horowitz, E. and Sahni, S., "Exact and Approximate Algorithms for Scheduling Nonidentical Processors," *Journal of ACM,* Vol. 23, No. 2, April 1976, pp. 317-327.

Hu, T. C., "Parellel Sequencing and Assembly Line Problem," *Operations Research,* Vol. 9, Nov. 1961, pp. 841-848.

Karp, R., "Reducibility among Combinatorial Problems," *Complexity of Computer Computations,* R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85-104.

Kasahara, H. and Narita, S., "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. Computers,* Vol. C-33, No. 11, Nov. 1984, pp. 1023-1029.

Kohler, W. H., "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Trans. Computers,* Vol. 24, 1975, pp. 1235-1238.

Korn, J., Papastavrou, J., James, R. M., and Pattipati, K. R., "Coordinated Many-on-Many Weapon Assignment Algorithms," *TR-306, Alphatech, Inc., Burlington MA,* Sept. 1986.

Kurien, T., Allen, T. G., Washburn, R. B., and Jr., "Parallelism in Multi-target Tracking and Adaption to Multi-processor Architectures," *Proceedings of the* 9th *MIT/ONR Workshop on* C^3 *Systems,* December, 1986, pp. 45-51.

Lavenberg, S. S., "Computer Performance Modeling Handbook," *Academic Press, Inc.,* 1983.

Lee, S. Y. and Aggarwal, J. K., "A Mapping Strategy for Parallel Processing," *IEEE Transactions on Computers,* Vol. C-36, No. 4, April 1987, pp. 433-442.

Loulou, R., "Tight Bounds and Probabilistic Analysis of Two Heuristics for Parallel Processor Scheduling," *Math. Oper. Res.*, Vol 9, 1984, pp. 142-150.

Ma, P., "A Model to Solve Timing-critical Application Problems in Distributed Computer Systems," *IEEE Computer,* Vol. 17, No.1, Jan. 1984, pp. 62-68.

Ma, P., Lee, E., and Tsuchiya, M., "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. Computers,* Vol. C-31, No.1, Jan. 1982, pp. 41-47.

Nillson, N., "Principles of Artificial Intelligence," *Pato Alto,* Calif. Tiogo, 1980.

Pattipati, K. R., Kastner, M. P., Dunham, S. R., Teele, J. L., Decker,J. C., "Fault-tolerant Computer Architecture Modeling and Analysis," *TR-305, ALPHATECH, Inc.*, November, 1986.

Pearl, J., "Heuristics," *Addison-Wesley,* 1984.

Polychronnopoulos, C. D. and Banerjee, U., "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds," *IEEE Transactions on Computers,* Vol. C-36, No. 4, April 1987, pp. 410-420.

Sahni, S. K., "Algorithms for Scheduling Independent Tasks," Journal of ACM, Vol. 23, No. 1, Jan. 1976, pp. 116-127.

Sethi, R., "Scheduling Graphs on Two Processors," *SIAM J. Computing,* Vol. 5, 1975, pp. 73-82.

Shen, C. C. and Tsai, W. H., "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing System Using a Minmax Criterion," *IEEE Trans. Computers,* Vol. C-34, No. 3, Mar. 1985, pp. 197-203.

Stone, H. S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

Syslo, M. M., Deo N., and Kowalik, J. S., "Discrete Optimization Algorithms with Pascal Programs," *Prentice Hall, Englewood Cliffs,* NJ, 1983.

Ullman, J. D., "Polynomial Complete Scheduling Algorithms," *Journal of Computer Science,* Vol. 10., No. 3, Jan. 1975, pp. 384-393.