1992-09

Allocation of periodic tasks with precedences on transputer-based systems

Falcao, Marco A. G.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/23524

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (If applicable) 33		7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School		
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000			7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code)			10. SOURCE OF FUNDING NUMBERS		
			Program Element No.	Project No.	Task No.	Work Unit Accession Number

11. TITLE (Include Security Classification)
ALLOCATION OF PERIODIC TASKS WITH PRECEDENCES ON TRANSPUTER-BASED SYSTEMS

12. PERSONAL AUTHOR(S) Falcao, Marco A. G.			
13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED From To	14. DATE OF REPORT (year, month, day) September 1992	15. PAGE COUNT 199

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUBGROUP	ADA, Allocation, Multicomputers, Multitasking, Task, Transputers

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

Task allocation is an important component of the process of mapping modules of application programs to multicomputers. A scheme for static allocation of periodic tasks with precedences to processors is developed considering task execution times, communication costs, and utilization level of each processor. It has the main goal of minimizing the application response time with a minimum number of processors.

A network of transputers is employed as a platform to experimentally evaluate the allocation approach constructed with this work. An existing communication layer in the language ADA is improved to provide an efficient support for task flow simulations on transputer networks.

The first phase of the allocation scheme is a constructive assignment heuristic that allocates the cluster of tasks composed of all tasks in the critical path to the same processor. The remaining tasks are allocated according to a heuristic function that considers task precedences, task execution times, and relative sizes of intertask messages.

The initial allocation is improved in the second phase by using an iterative pairwise interchange of tasks that considers interprocessor communication distances.

The overall scheme of task allocation was successfully tested and analyzed through simulation of several applications on a transputer network providing a near optimal solution.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT		21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS		UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Shridhar B. Shukla	22b. TELEPHONE (Include Area code) (408) 6-46-2764	22c. OFFICE SYMBOL EC/Sh

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted   SECURITY CLASSIFICATION OF THIS PAGE
All other editions are obsolete   UNCLASSIFIED

Allocation of Periodic Tasks
with Precedences
on Transputer-based Systems

by

Marco A. G. Falcao
Lieutenant Commander, Brazilian Navy
B.S., University of Sao Paulo

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
from the

NAVAL POSTGRADUATE SCHOOL
September 1992

ABSTRACT

Task allocation is an important component of the process of mapping modules of application programs to multicomputers. A scheme for static allocation of periodic tasks with precedences to processors is developed considering task execution times, communication costs, and utilization level of each processor. It has the main goal of minimizing the application response time with a minimum number of processors.

A network of transputers is employed as a platform to experimentally evaluate the allocation approach constructed with this work. An existing communication layer in the language ADA is improved to provide an efficient support for task flow simulations on transputer networks.

The first phase of the allocation scheme is a constructive assignment heuristic that allocates the cluster of tasks composed of all tasks in the critical path to the same processor. The remaining tasks are allocated according to a heuristic function that considers task precedences, task execution times, and relative sizes of intertask messages.

The initial allocation is improved in the second phase by using an iterative pairwise interchange of tasks that considers interprocessor communication distances.

The overall scheme of task allocation was successfully tested and analyzed through simulation of several applications on a transputer network providing a near optimal solution.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. ELEMENTS OF PROGRAM EXECUTION ON MULTICOMPUTERS

Since the advent of very large-scale integrated (VLSI) circuits, computer hardware has decreased in size and cost. In addition, the employment of higher-performance parallel computers has become essential for several modern applications such as weather prediction, computational aerodynamics, artificial intelligence, and military command and control systems. Therefore, new software tools are required to exploit the parallelism on these machines as well as to provide transparent program development systems to the users.

Concurrent computing using multicomputers requires a mapping between the user application and the processing nodes. This mapping procedure is usually decomposed in distinct elements to make this complex problem tractable. These elements are partitioning, task allocation, node scheduling, and message routing, as shown in Figure 1.

Partitioning is the part that divides the original problem into subproblems that are solved by individual processors. The solutions of these subproblems are combined to compose an overall solution to the original problem. Therefore, it must determine the computation grain size, which is the number of operations performed in a task between intertask

1

communication, and the communication grain size, which
corresponds to the message size.
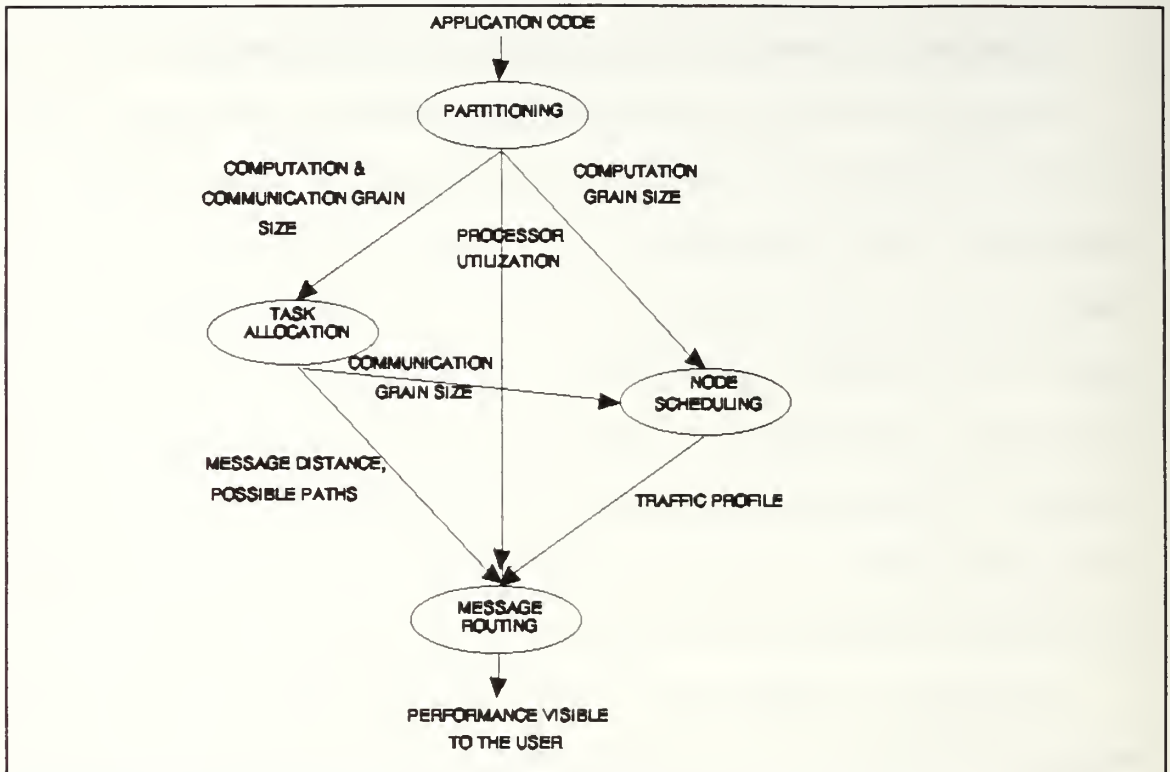
Figure 1: Interaction between stages of the mapping procedure

Task allocation is the component that allocates the pieces
of program (program modules) obtained from partitioning to the
processors that compose the parallel computer. Locations of
the source and destination of messages, which constrain the
possible shortest paths used in message routing, are
established by this element.

Node scheduling considers task features such as
precedence, execution time and deadline, in order to schedule
task executions on each node processor. Thus, it affects the

2

traffic profile by determining the instants of message transmission from one node processor to another.

Message routing decides how effectively the communication subsystem can be implemented. It considers the minimization of network latency, uniform utilization of the network resources, and freedom from deadlock/livelock.

B. REQUIREMENTS IMPOSED BY APPLICATIONS

Real-time applications impose a diversity of requirements that must be considered in the design of the new generation of parallel computers. These requirements are not constant in time but they can change frequently. For instance, consider the problem of scheduling aperiodic tasks with time constraints and random interval of arriving, or the problem of meeting new system specifications to enhance the original system[STANKOVIC 88]. Therefore, the computer power needed for such applications as measured in terms of MIPS or MFLOPS is not static and should be provided on demand.

Since the design of new applications implies high development costs, it is highly desirable that new systems should be dynamic and flexible. Therefore, system designers must consider approaches to minimize the number of constraints to be imposed on further modifications. For instance, this can be accomplished by using scalable parallel computer architectures and real-time operating systems with support for dynamic allocation of tasks.

Another fundamental aspect of real-time applications is that the correctness of system results is not only dependent on the logical correctness of the program but also on its ability to meet the individual deadlines imposed by individual tasks. Consequently, critical tasks should be scheduled for execution with higher priority than non-critical tasks. Critical tasks are processes for which the consequence of missing a deadline is catastrophic with violation of personal or material safety.

Besides deadlines, task execution times, repetition rates of periodic tasks, and precedence between tasks are also very important requirements.

Fault tolerance is another issue that must be considered. Multiple instances of the same process must run on different processors to achieve the desired reliability.

Resource constraints are imposed by the system architecture. Therefore, tasks are required to share several resources other than CPU such as I/O channels, data structures, files, and databases.

Basic to such sharing is intertask communication that may result in interprocessor communication. The presence of an efficient communication system underlying a multicomputer network is a requirement that must be satisfied to minimize the effects of delays due communication between tasks residing at different nodes. These delays are the cause of the saturation effect on the achievable speed-up when using

multiple processors. In addition, they are not predictable causing problems for real-time applications.

Static systems are systems in which tasks and timing constraints such as the arrival times of tasks are known beforehand. Dynamic systems are those systems which tasks arrive at random times. A static resource allocation scheme is required for static systems. This approach can be employed at compile-time since all needed information is known then. On the other hand, a dynamic resource allocation scheme is required for dynamic systems since it has to be used at execution time and little is known about the application at compile-time.

C. CURRENT STATE OF THE ART

There have been many studies in static scheduling in distributed systems. This category of scheduling problems has been traditionally formulated as the task allocation problem. A useful survey on task allocation problems can be found on [STANKOVIC 87].

Several approaches to the allocation model have been used. They are graph theoretical, integer programming, and heuristic methods[TSUCHYA 82]. These approaches pursue solutions that minimize the interprocessor communication costs with a balanced utilization of each processor.

The scheme presented by [TSUCHYA 82] employs integer programming models to find optimal allocation with explicit

5

time constraints. A branch and bound search algorithm is used to solve the allocation problem. This approach can be used to balance processor loads and to minimize communication costs. An AND-OR precedence graph with 23 tasks having port-to-port timing requirements is used to validate this scheme. This model allows the inclusion of several constraints such as allocation preference, task exclusion, redundancy, time and resource constraints.

The approach developed by [RAMAMRITHAM 89] uses a heuristic-directed search. This algorithm handles periodicity constraints, precedence, communication, and fault tolerance. Experimental evaluation of the algorithm shows that a task set can be feasibly allocated and scheduled, and the algorithm is highly likely to find it without any backtracking during the search.

D. OBJECTIVES

The objective of this thesis is to develop suitable algorithms for static allocation of tasks having precedence constraints on transputer-based systems. Specialized research in this field has shown that optimal solutions for this problem are NP-hard[QUINN 87], therefore heuristic methods are required to find near optimal solutions.

Several task flows are employed on a transputer-based system to evaluate the performance of the allocation algorithms with respect to the interprocessor communication

costs, load balance on each processor and task flow response time.

An existing communication layer software, developed by [RICHMOND 91] in ADA, is upgraded to implement task flow simulations on a transputer-based system. Modifications to the communication layer that were necessary to improve its functionality are as follows:

- Take similar communication routines existing on each node main program and put them into a single ADA package to make them easy to be reused as well as to improve the total compilation time of the distributed program.

- Use different approaches to route messages among the nodes that compose the transputer network. The first implementation routes messages in a counter clockwise ring fashion only.

- Reduce communication overhead in relation to the total execution time of the distributed program.

- Study how to make the layer more robust with respect to deadlocks.

A network of transputers is used in this thesis for the simulation because it follows MIMD(Multiple Instruction Stream Multiple Data Stream) parallel architecture providing flexibility for further topological changes as well as scalability. Moreover, its use makes the simulations more realistic and simpler than purely theoretical models that could be built in uniprocessor computers.

The programming language ADA is used in the current thesis due the following reasons:

- Existence of a communication layer written in ADA [RICHMOND 91] that makes the communication between tasks location invariant, and

- Presence of high level ADA constructs that make it easier to implement multitasking. ADA has incorporated a technique called the Rendezvous[HOARE 78] that combines mutual exclusion, task synchronization, and interprocess communication.

- The language is a DOD standard(ANSI/MIL-STD-1815A-1983) and has the potential to be the standard programming language in the next generation of real-time systems.

E. THESIS ORGANIZATION

This thesis contains four chapters. Chapter I described the elements of program execution on multicomputers, requirements imposed by applications on real-time parallel computer systems, a brief overview of the current state of the art on the problem of task allocation, and the objectives of the present work.

Chapter II contains the general structure of the communication layer including discussions about topics such as the AUV flow simulation, deadlock avoidance, characterization

of the communication performance, and changes to meet new project specifications.

Chapter III describes the task allocation including the constructive assignment heuristic, the iterative improvement by pair-wise interchange of tasks, and simulations on the transputer network.

Chapter IV states the conclusions and recommendations for further research.

II. COMMUNICATION LAYER

A. GENERAL STRUCTURE

1. OCCAM HARNESSES MODIFICATION

Figure 2 shows the T-800 transputers that compose the transputer network used in this thesis. Processor EARTH is linked directly to the host PC, and therefore it is the only processor with access to the I/O offered by the PC keyboard and display. All other processors(MARS, VENUS, PLUTO and SATURN) can only communicate with the host PC by sending messages to the processor EARTH. Task SCREEN that runs on EARTH can inspect the codes of messages sent by these processors and then print suitable messages on the PC screen.
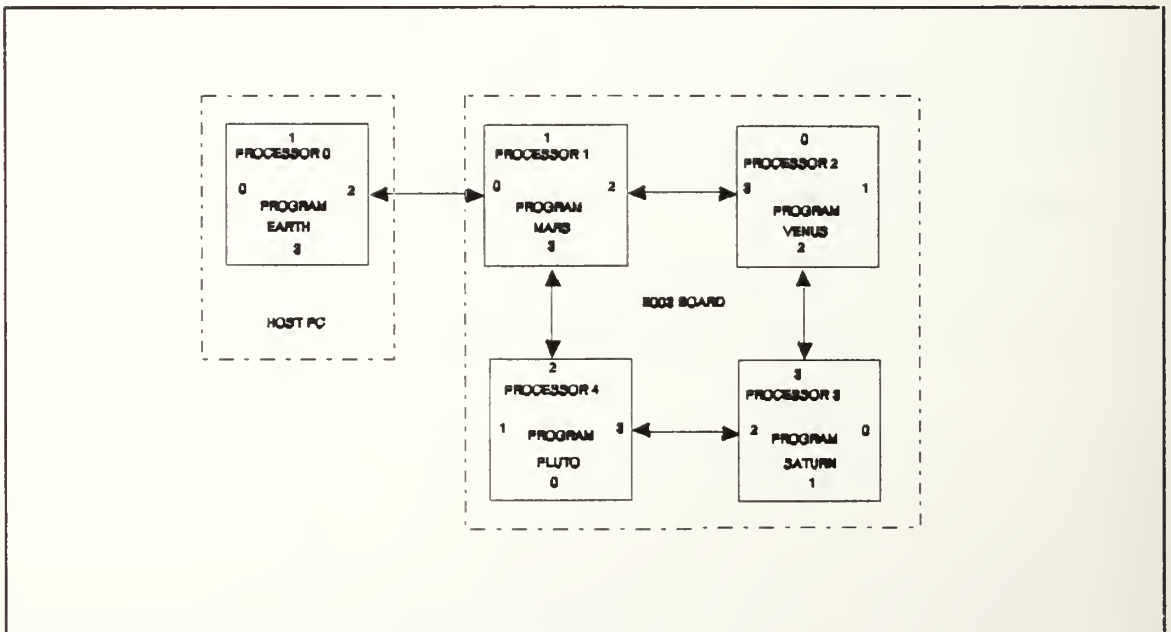
Figure 2: Communication topology

Processors MARS, VENUS, PLUTO and SATURN are the transputers that run the distributed software for the application task flow simulation. The OCCAM harnesses for processors MARS, VENUS, PLUTO and SATURN have been modified to permit the use of different approaches for routing messages through the transputer network.

Tables I, II, III, and IV show how the communication channels are allocated on the processors MARS, PLUTO, SATURN and VENUS respectively. The channel numbers shown in Figure 2 are the physical link numbers.

Table I: ALLOCATION OF CHANNELS ON MARS

IN/OUT CHANNEL	NAME	VIRTUAL NUMBER	PHYSICAL NUMBER
INPUT	Earth2Mars	2	0
INPUT	Venus2Mars	4	●
INPUT	Pluto2Mars	5	●
OUTPUT	Mars2Earth	2	0
OUTPUT	Mars2Venus	4	2
OUTPUT	Mars2Pluto	5	3

Table II: ALLOCATION OF CHANNELS ON PLUTO

IN/OUT CHANNEL	NAME	VIRTUAL NUMBER	PHYSICAL NUMBER
INPUT	Mars2Pluto	3	●
INPUT	Sat2Pluto	4	3
OUTPUT	Pluto2Mars	3	2
OUTPUT	Pluto2Sat	4	3

11

Table III: ALLOCATION OF CHANNELS ON SATURN

IN/OUT CHANNEL	NAME	VIRTUAL NUMBER	PHYSICAL NUMBER
INPUT	Pluto2Sat	2	2
INPUT	Venus2Sat	3	3
OUTPUT	Sat2Pluto	2	2
OUTPUT	Sat2Venus	3	3

Table IV: ALLOCATION OF CHANNELS ON VENUS

IN/OUT CHANNEL	NAME	VIRTUAL NUMBER	PHYSICAL NUMBER
INPUT	Mars2Venus	2	3
INPUT	Sat2Venus	5	2
OUTPUT	Venus2Mars	2	3
OUTPUT	Sat2Venus	5	2

The physical link numbers are associated with the hardware communication channels existent on each transputer. The T800 transputer has 4 bi-directional communication channels that are numbered 0, 1, 2 and 3. The virtual channel numbers are logical references to the communication channels declared on each main ADA program.

The Alsys ADA compiler for transputer networks requires that each main ADA program to be run in a transputer node must have its own OCCAM harness in order to provide a clean interface to the program in terms of used channels[ALSYS

90]. Tables V, VI, VII, VIII, and IX show all OCCAM programs
that are required for each processor node to run application
task flows using a distributed ADA program on the transputer
network shown in Figure 2.

Table V: OCCAM HARNESSES ON PROCESSOR EARTH

PROGRAM	FUNCTION
EARTHH.OCC (EARTH HARNESS)	Declares the virtual channels used by EARTH and invokes the EARTH ADA program
EARTHH2.OCC	Declares the entry point that is used by EARTH harness to call the EARTH ADA program
MERGER.OCC	Collect output errors from up to 20 Ada programs
MAINH.OCC	Invokes the EARTH harness and the MERGER error harness
MAIN.PGM	Maps the virtual channels of each processor to physical links. Invokes the following harnesses in parallel: MAIN, MARS, PLUTO, SATURN and VENUS

The OCCAM harnesses which declare the virtual channels
described in Tables I, II, III and IV, and the OCCAM program
that allocates virtual channels to physical links(MAIN.PGM)
are enclosed in Appendix A. They have been modified to permit
bi-directional communications between pairs of adjacent
transputers. The previous implementation sends messages in one
fixed direction around the ring. Different schemes of routing

13

Table VI: OCCAM HARNESSES ON PROCESSOR MARS

PROGRAM	FUNCTION
MARSH.OCC (MARS HARNESS)	Declares the virtual channels used by MARS and invokes the MARS ADA program
MARSH2.OCC	Declares the entry point that is used by MARS harness to call the MARS ADA program

Table VII: OCCAM HARNESSES ON PROCESSOR PLUTO

PROGRAM	FUNCTION
PLUTOH.OCC (PLUTO HARNESS)	Declares the virtual channels used by PLUTO and invokes the PLUTO ADA program
PLUTOH2.OCC	Declares the entry point that is used by PLUTO harness to call the PLUTO ADA program

messages can be used with the new version.

Table VIII: OCCAM HARNESSES ON PROCESSOR SATURN

PROGRAM	FUNCTION
SATURNH.OCC (SATURN HARNESS)	Declares the virtual channels used by SATURN and invokes the SATURN ADA program
SATURNH2.OCC	Declares the entry point that is used by SATURN harness to call the SATURN ADA program

Table IX: OCCAM HARNESSES ON PROCESSOR VENUS

PROGRAM	FUNCTION
VENUSH.OCC (VENUS HARNESS)	Declares the virtual channels used by VENUS and invokes the VENUS ADA program
VENUSH2.OCC	Declares the entry point that is used by VENUS harness to call the VENUS ADA program

2. THE HOST COMMUNICATION LAYER PACKAGE

The host communication layer is the software that manages the communication of the host transputer(EARTH) with other nodes. This layer is implemented by the generic package HOST_LAYER enclosed in Appendix B.

The package HOST_LAYER includes only one task, the task EARTH_QUE, which implements a circular buffer that is used to keep the incoming messages until they can be sent to their destination. This task works like a local mailman that

15

is in charge of storing and delivering the messages to the local tasks. Further details of this task will be given later since its implementation is identical to task QUE of package COMLAYER.

An instance of the package HOST_LAYER must be generated with the procedure SEND_IT, which is the program that issues entry calls to the local tasks passing the messages to their destinations. The procedure SEND_IT is called by task EARTH_QUE.

An instance of the package HOST_LAYER is created in the main program EARTH.ADA, which is also enclosed in Appendix B, with the procedure SEND_IT_FROM_EARTH. This procedure sends messages to the local task existent on EARTH.ADA that is SCREEN.

Task SCREEN can be used for two main purposes:

- Allow non host transputer nodes(MARS, PLUTO, SATURN, VENUS) to send output messages to the host PC screen. This can be done by setting an appropriate message code that can be interpreted by task SCREEN that can print the desired output as a result.

- Accumulate statistics about the AUV flow simulation. For instance, task SCREEN accumulates the total execution time and the number of iterations of task VEHICLE_SYS that can be used to calculate the average loop execution time.

Figure 3 illustrates the use of the HOST_LAYER by the program EARTH.ADA. The messages are read from the input channel(Mars2Earth) by the main program and sent to the circular buffer implemented by task EARTH_QUE. From there the messages are sent through procedure SEND_IT_FROM_EARTH to task SCREEN that outputs information to the PC screen.

3. THE COMMUNICATION LAYER PACKAGE

The communication layer is the software that manages the communication with external nodes by non host transputers(MARS, PLUTO, SATURN and VENUS). This layer makes implementation of Ada multitasking in the transputer network to be location invariant because messages are sent from task to task by using the same communication primitives independent of the task location in the transputer network. It is implemented by the generic package COMLAYER enclosed in Appendix B. It should be noticed that, since the package HOST_LAYER does not have a message traffic handler like task INOUT on package COMLAYER, none of the application tasks should execute on EARTH.

The package COMLAYER encapsulates tasks INOUT and QUE. These tasks are described as follows:

a. TASK INOUT

This task accepts messages from either the main program(external messages arriving at a node) or from local tasks (messages going to other nodes or even messages going to

17
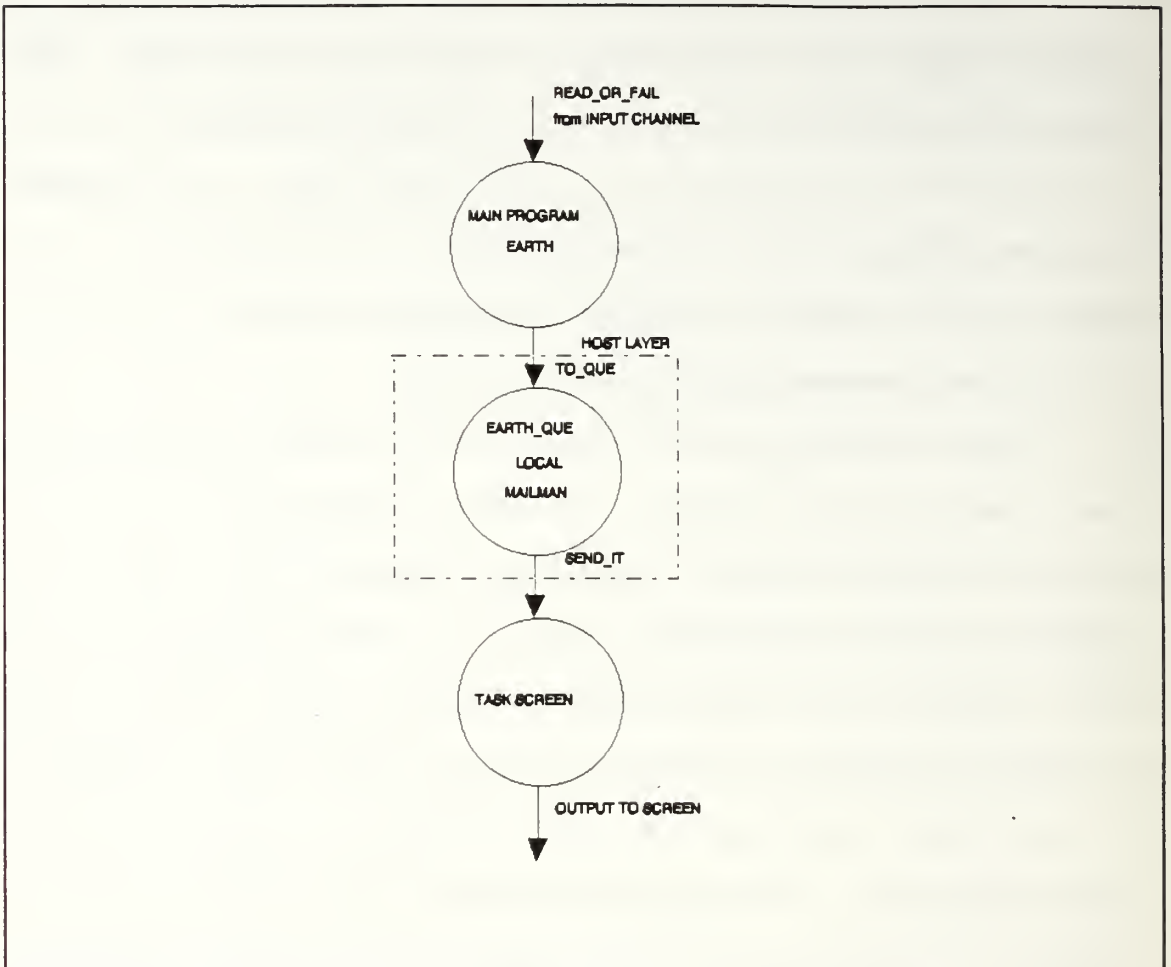
Figure 3: Host layer used by EARTH

other tasks in the same node). The function WHERE_IS that is
declared in package COMMON, enclosed in Appendix B, is called
and returns the destination node of the message. If the
destination node is the local node the message will be sent to
the task QUE that is responsible for the local message
distribution. On the other hand, if the message destination is
an external node it will be sent to this node through a
communication channel. The INOUT task also chooses the

18

communication channel to be used depending on the routing strategy used for the network communication.

There are four possible routing strategies that can be used with the network topology shown in Figure 2 that are declared in the package COMMON:

- Counter-clockwise ring direction(CNT_CLK_RING) - This is the approach that was used in the first implementation. The messages are always sent in the direction MARS->PLUTO->SATURN->VENUS->MARS.

- Clockwise-ring direction(CLK_RING) - The messages are always sent in the direction MARS->VENUS->SATURN->PLUTO->MARS.

- Multipath(MULTI_PATH) - Chooses one output channel at random among all possible lower cost output channels to a destination processor. The package RANDOM, enclosed in the Appendix B, was extracted from [ALSYS 90] to generate random numbers. This routing strategy is based on the multipath routing technique for a store-and-forward network layer implementation and is described in [TANENBAUM 89]. The lower cost output channels were obtained by inspection of Figure 2 taking into account the number of hops between the nodes. The shortest path algorithm due to [DIJKSTRA 59] can be used to compute the shortest path between two nodes for more complicated networks. This algorithm is described in [TANENBAUN 89].

19

- Best path(BEST_PATH) - This technique is suitable for hypercube networks. The network formed by the nodes MARS, VENUS, SATURN and PLUTO in Figure 2 is a simple hypercube of order 2. The algorithm always chooses the output channel that eliminates deadlock among the lower cost channels. This routing approach is based on algorithm A described in [TZENG 91]. The output channel is chosen by taking into account the numbers that identify the processors(node addresses) in a hypercube. The relative address of the message source node to its destination node(the relative address of two nodes, say x and y, is the bitwise Exclusive-or of their addresses, x XOR y) is employed to choose the best channel to send the message. This algorithm eliminates deadlock by avoiding the formation of cycles by the routing paths.

The package COMMON declares all routing tables that are used by these routing techniques. These tables must be generated by dedicated programs using the four approaches mentioned. It can be noticed that the routing tables are highly dependent on the topology used. Therefore, they must be recalculated whenever there is a change in the topology. Some of them are even useless for certain topologies. For instance, it does not make sense to use a ring for an hypercube of order greater than two or for a mesh with a larger number of transputers.

Looking forward to further expansion of the employed network, it can be noticed that MULTIPATH seems to be

the best alternative for MESH topologies while BEST_PATH seems to be the best for HYPERCUBE topologies.

Task INOUT needs to be initialized with the following data:

- SITE - Informs the identification of the current node processor.

- SEND_ARRAY - Array of type CHANNELS.CHANNEL_ARRAY that must contain all output nodes used by this node processor. For each transputer node we can have up to 4 output channels that are declared in the node processor main program. The package CHANNELS contains all communication primitives that are offered by the Alsys Ada compiler system to run a multitransputer program[ALSYS 90].

- SEND_TABLE - Table of type OUT_TABLE. The type OUT_TABLE is defined in the package COMMON. It is an unconstrained array(PROGRAMS range<>, NATURAL range<>) of BOOLEAN. The PROGRAMS range is constrained to the range of node programs MARS..PLUTO(MARS, VENUS, SATURN and PLUTO). The NATURAL range is constrained to the number of output channels. The BOOLEAN content of SEND_TABLE(DESTINATION_PROGRAM_INDEX, OUTPUT_CHANNEL_INDEX) has the following meaning:

.TRUE - OUTPUT_CHANNEL_INDEX points to a valid output channel that may be used for sending messages to the destination program pointed by the DESTINATION_PROGRAM_INDEX.

.FALSE - The output channel pointed by OUTPUT_CHANNEL_INDEX is not a valid channel to be used for

21

sending messages to the destination program pointed by the DESTINATION_PROGRAM_INDEX.

The main entry points defined for task INOUT can be seen as communication primitives offered to the main node programs and to the tasks belonging to them to make multitasking in a transputer network location invariant. The mentioned entry points are described as follows:

- INCOMING - It is called by the main node program to pass a message just arrived from an external node.

- SEND - It is used by local tasks either to send messages to remote tasks(tasks located in external nodes) or to send messages to other local tasks. It is emphasized here that the local task does not have to worry about where the destination task is located and this feature makes the task's communication location invariant. Communication transparency is provided because the same primitives are used to express both remote and local communications.

b. *TASK QUE*

This task implements a kind of QUEUE that is a circular buffer. It is called from task INOUT that passes the message to be distributed locally. This message is placed in the circular buffer and waits for its time of delivery. The messages are delivered in order of arrival but, if a rendezvous with the receiving task cannot be established, this message loses its opportunity to be delivered and has to wait

22

for the next circular buffer scan to have another opportunity. Therefore, the order of delivery of the messages is not guaranteed to be the same as the order of arrival in the circular buffer.

An instance of the procedure that issues entry calls to the local tasks(SEND_IT) must be created on each processor node. This procedure is not transparent in relation to the allocation of tasks on the transputers. This happens because the user should know which tasks are allocated to each transputer to write each specific node procedure to generate SEND_IT.

Figure 4 shows the relationship between the tasks in one node program when using the communication layer. It can be noticed that the task WAITING that was used in the first implementation[RICHMOND 91] was eliminated from the communication layer. Task WAITING uses one additional queue that may be employed between INOUT and QUE tasks and was eliminated because it was not really used by the communication layer.

4. ADA IN A DISTRIBUTED ENVIRONMENT

Several approaches can be applied for implementation of the ADA language in distributed systems. These approaches are reviewed by [ATKINSON 88] and the most important are summarized as follows:
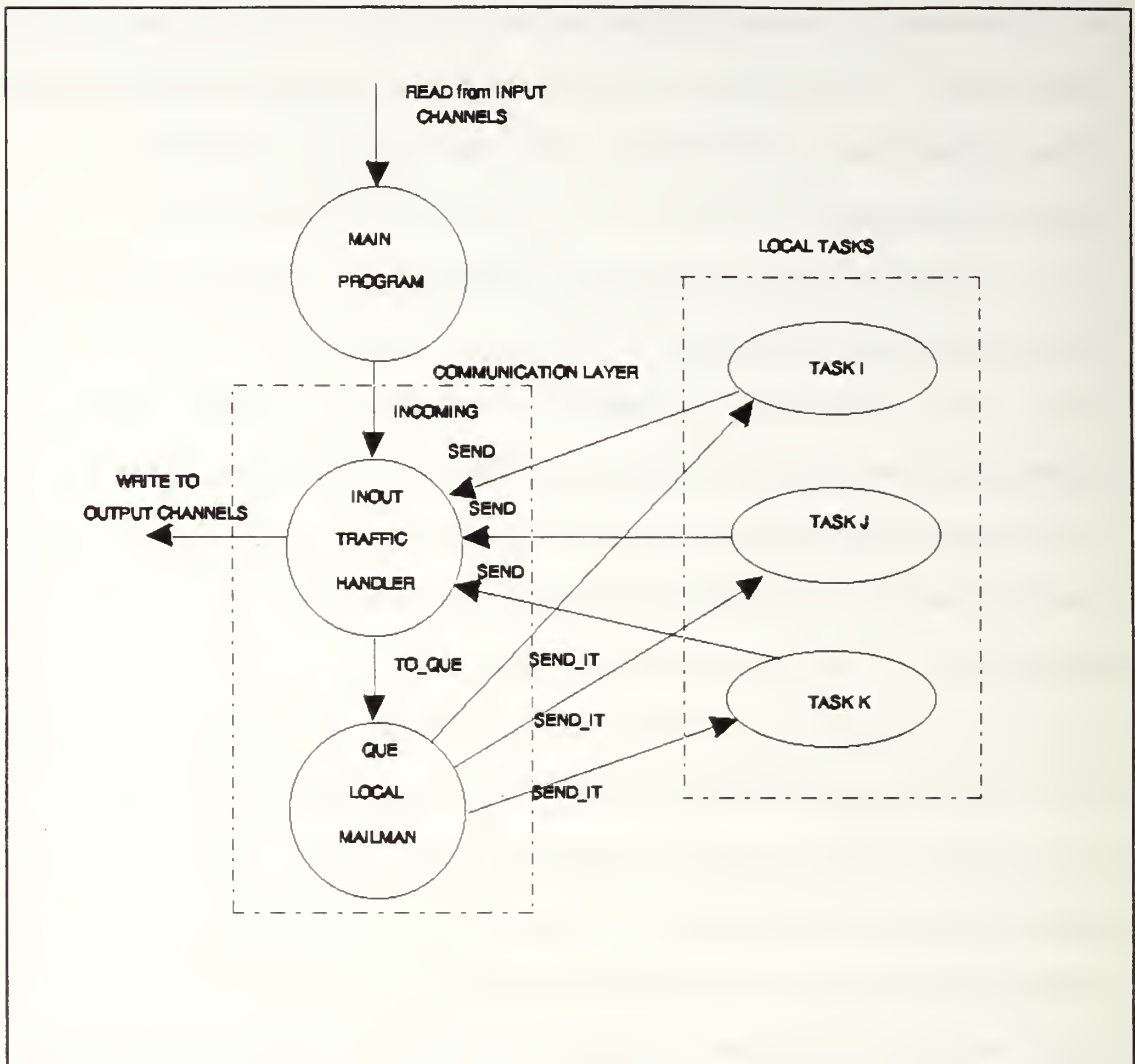
Figure 4: Communication layer relation to the main program

- Full ADA, Multiple Communicating Programs - In this case one separate ADA program is required for each processor node that composes the transputer network. These programs use basic communication primitives to communicate with each other. This requires knowledge of the underlying network by the programmer but leads to a faster implementation. The main disadvantages of this approach are that programs must be

24

rewritten when the network is changed and that ADA semantics regarding multitasking is not fully implemented.

- Full ADA, Single Program - The program is written as only one ADA program. This approach fully implements the ADA semantics and, as a consequence, it is highly portable. The problems of partitioning, task allocation and task distribution must be solved by the compiler. Unfortunately, compiler technology for distributed programs has not solved these problems in a satisfactory way. In addition, it requires a fairly complicated communication software system to support the full ADA semantics across the network.

- Non-Uniformly Restricted ADA, Single Program - In this case, the ADA semantics is fully implemented within the limits of abstract entities called virtual nodes. Communication across the boundaries of the virtual nodes is achieved by remote rendezvous.

The implementation of the communication layer employs a solution based on the Full ADA Multiple Communicating Programs approach because it employs one ADA program for each node processor. In addition, communication between these programs is achieved by basic primitives offered by the package CHANNELS.

In the first implementation of the communication layer, the remote rendezvous is simulated by task QUE at the receiving node(CALLEE) that sends an acknowledge message to the task origin (CALLER). The origin of the message will not

proceed until it gets the acknowledge message from the CALLEE. In fact, this is the only way to go when working with the following types of entry calls[ATKINSON 88]:

- Entry call that has out parameters. It works as a remote subroutine call so that the CALLER has to wait for the CALLEE return before proceeding.

- Conditional entry calls - A conditional entry call issues an entry call that is then canceled if a rendezvous is not immediately possible[VOLZ 85]. Due the time delays in the remote communication the CALLER may cancel an entry call when the CALLEE is still able to accept the entry call.

- Timed entry calls - A timed entry call issues an entry call that is canceled if a rendezvous is not started within a given delay[REYNOLDS 83]. There is a problem of interpretation in where to consider the time delay. It can be considered from the CALLER or from the CALLEE time reference.

The acknowledge messages employed in the first version of the communication layer were avoided in the new version due to the following reasons:

- These messages may lead to a deadlock situation in case of using MULTIPATH or BEST_PATH routing strategies. This problem seems to be related with the cycle formed by tasks INOUT and QUE when INOUT may send a message to QUE at the same time that QUE is sending an acknowledgement to INOUT.

- The AUV tasks can communicate by using simple messages without requiring an immediate return from the

destination. Timed entry calls are issued by task QUE in the CALLEE node. Therefore, it does not seem to be a big problem if the CALLER task proceeds before the rendezvous with the CALLEE has been established. After all, if the CALLER task depends on results of an action to be taken by the CALLEE task, it will suspend in an accept statement waiting for the required results. Thus, the new version of the communication layer does not implement the Ada rendezvous in a peer to peer sense but rendezvous is used between the tasks that implement the communication layer and the local node tasks.

- The acknowledge messages contributed to the total overhead of the communication layer. Since for every message there is an associated acknowledge message, it practically doubles the communication load.

5.   DEADLOCK AVOIDANCE

There are at least four potential deadlock situations that may occur in distributed systems leading to a general blocking situation:

- Asynchronous use of the communication channels - For instance, in Figure 2, consider the case where VENUS attempts to send a message to MARS at the same time that MARS attempts to send a message to VENUS. Both processors are using the shortest path links to send their messages. Both processors are using the primitive WRITE of the package CHANNELS that is not preemptive. Therefore, if none of the transputers gives

up, then the programs will deadlock. If this problem happens with the communication layer, the situation may be recovered because the WRITE primitive is called from task INOUT and the READ primitive is called from the MAIN program on each node. Since the tasks are switched on each processor node, it is possible that the WRITE primitive in a node could get synchronized with a READ primitive of the other node due the random nature of the task activation mechanism, however, this is not guaranteed. The program user should then enforce the synchronization when using the primitives READ and WRITE to ensure that this situation will never happen. In this communication layer absence of deadlock is guaranteed by using Best Path or Ring routing strategies. The Multipath routing strategy is not guaranteed to be deadlock-free due its random mechanism. Another solution is to use the READ_OR_FAIL and WRITE_OR_FAIL primitives of the package CHANNELS. These primitives employ a preemptive time-out that gives up the use of the primitive if it cannot succeed within a specified period. This solution may cause indefinite postponement that is also undesirable.

- Lack of the circular buffer space - For instance, if MARS is trying to send a message to PLUTO but this processor does not have room in its circular buffer to hold the new message, then PLUTO tries to send a message to another processor to free space for the incoming message. Considering that all other processors are also with their circular buffers

28

full, a deadlock situation will occur. This problem is avoided by choosing a suitable size for the circular buffer in task QUE of the communication layer to accommodate the traffic generated by the flow simulation. It can be noticed that an excessive size of the circular buffer implies waste of memory space as well as higher overhead for the communication layer[RICHMOND 91]. A circular buffer of size 20 was enough for all application flow simulations in this thesis.

- Asynchronism between tasks - Considering two tasks A and B, if task A issues an entry call to task B at the same time that task B issues an entry call to task A, deadlock situation will occur. One way to prevent this kind of deadlock is to avoid cycles in the directed graph that represents the task flow to be simulated. For the cases where cycles cannot be avoided it is required that all tasks in the cycle get their inputs before trying to send their outputs to obtain an overall cycle synchronization. Another alternative is to use preemptive entry calls(conditional or timed entry calls) for the rendezvous implementation. This solution has the inconvenience of indefinite postponement. Therefore, the programmer should enforce task synchronization to avoid this kind of deadlock.

- Asynchronism of messages arriving in one task - One single task may have several entries accepting messages from different tasks. These messages need to be synchronized to avoid potential deadlocks due different message arrival

times. Programs MARS, PLUTO, SATURN and VENUS use selective non-ordered loops to avoid this problem.

B. AUV SIMULATION FLOW
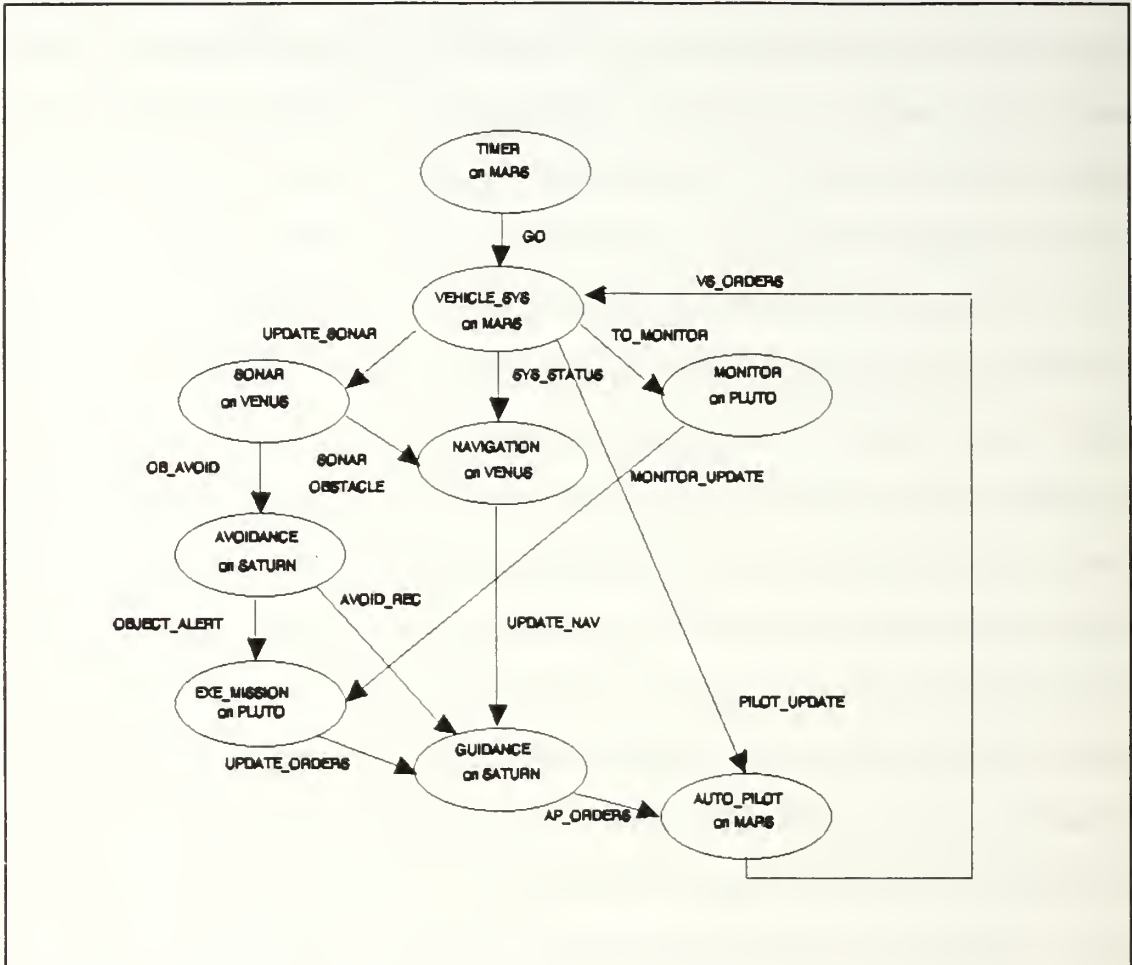
Figure 5: AUV simulation flow

Figure 5 shows the Autonomous Underwater Vehicle(AUV) simulation flow that was used in the original implementation and in the evaluation of the communication layer modification reported in this document. A detailed description of the AUV flow can be found in [RICHMOND 91].

Table X contains the execution times that are used for all tasks of the AUV flow simulation. Each value is only an estimated execution time based on the complexity of the component task and may not represent the real execution time. Task TIMER is not included in TABLE X because it is the task that controls the interval of repetition to be used by the AUV simulation with little overhead to the total execution time.

The AUV simulation flow is implemented by programs MARS.ADA, PLUTO.ADA, SATURN.ADA and VENUS.ADA, and the package

Table X: AUV FLOW - TASK EXECUTION TIMES

TASK	EXECUTION TIME(seconds)
VEHICLE_SYS	0.08
SONAR	0.02
NAVIGATION	0.05
MONITOR	0.03
AVOIDANCE	0.08
EXE_MISSION	0.06
GUIDANCE	0.07
AUTO_PILOT	0.04

COMMON.ADA. These programs are enclosed in Appendix B.

C.   DEADLOCK RELATED TO THE AUV SIMULATION FLOW

The following measures are used to avoid deadlocks when using the communication layer for the AUV implementation:

- The AUV flow shown in Figure 5 has multiple cycles and all of them start and finish at task VEHICLE_SYS. Due the data dependency to task AUTO_PILOT and the time control imposed by task TIMER, the task VEHICLE_SYS cannot issue entry calls until it accepts VS_ORDERS from AUTO_PILOT and GO from task TIMER. This feature ensures the task synchronization that avoids the deadlock.

- The acknowledge messages that were used in the original implementation were removed in the modified implementation because they were leading to deadlock for certain routing strategies.

- Task QUE employs a circular buffer to store the local messages and the procedure SEND_IT to distribute them to local tasks. An instance of the procedure SEND_IT is generated on each node. This procedure ensures a preemptive scheme of message delivery. The procedure issues an entry call, if the call cannot succeed within a given time delay, it is canceled and the message is skipped in the circular buffer. It has to wait for the next circular buffer scan to have another opportunity of being delivered. All instances of SEND_IT use zero delay for the timed entry call. This delay is important because it has the effect of suspending the task when the rendezvous with a local task cannot be established right away.

- Figure 4 shows another kind of cycle that is composed by tasks INOUT, QUE, and the local task. Each cycle is formed by the following task sequence: task INOUT -> task QUE -> local task -> task INOUT. These potential deadlocks are avoided because the local task only issues an entry call to INOUT after accepting all entry calls from task QUE.

D. CHARACTERIZATION OF COMMUNICATION PERFORMANCE

Figure 6 shows the Gantt Chart for the execution of the AUV flow of Figure 5 on the transputer network formed by processors MARS, PLUTO, SATURN and VENUS of Figure 2. This chart also shows the static task allocation that was used for the AUV flow simulation.

The static allocation is defined by the declaration of the type TASKS and the function WHERE_IS in package COMMON and by the instances of procedure SEND_IT on each node.

It can be seen from Figure 6 that the time spent by the AUV flow is lower bounded by 0.35 seconds that is the sum of the execution times of the tasks that compose the longest path from task VEHICLE_SYS in Figure 5(VEHICLE_SYS, SONAR, AVOIDANCE, EXE_MISSION, GUIDANCE and AUTO_PILOT). This minimum cost does not take into account the communication overhead.

Therefore, the maximum theoretical speed-up that can be achieved with this data-flow considering that it cannot be pipelined due to existent data dependencies is 0.43/0.35=1.23. The value of 0.43 seconds is obtained by adding all task

33
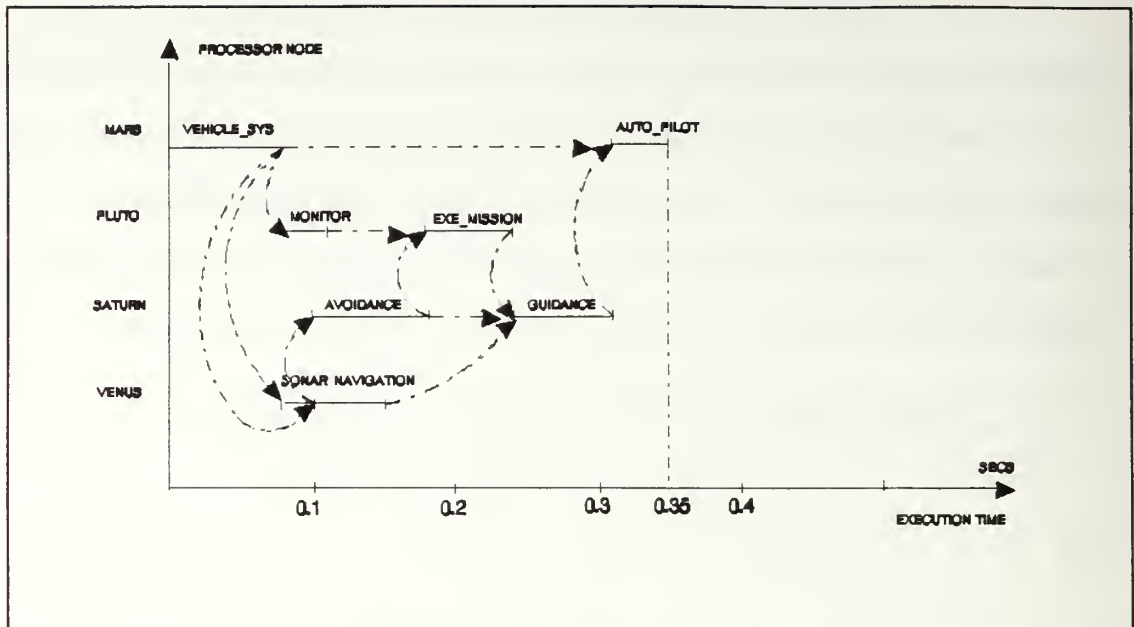
Figure 6: Gantt chart for the AUV flow execution

execution times giving a time that is equivalent to the time
needed to run the flow in a single processor.

Due to the AUV flow limitation, the best speed-up that can
be achieved is much lower when compared with the possible
linear speed-up for 4 processors(1.23 < 4). This happens
because the AUV flow is not parallel enough to exploit all
potential parallelism of the 4 transputers. This fact can also
be noticed by the low utilization of the processors as shown
in Figure 6.

Table XI shows the best results that were achieved for the
AUV flow execution time with different routing strategies.

From Table XI it can be noticed that the maximum
communication overhead obtained is 23.1 msec given by the
counter clockwise ring routing strategy(0.3731secs-
0.35secs=23.1 ms). The minimum communication overhead obtained

34

Table XI: EXPERIMENTAL EXECUTION TIMES FOR THE AUV FLOW

Routing strategy	Counter clock-wise ring	Clock-wise ring	Multi-path	Best path
Execution time(s)	0.3731	0.3693	0.3675	0.3675

is 17.5msec given by both multipath and best path routing strategies(0.3675secs-0.35secs=17.5 msec). Therefore, an improvement is achieved by using different routing strategies than the counter clockwise ring routing strategy even for transputer networks with very few components, as in the case of Figure 2. This difference tends to be bigger when using more complex transputer networks.

Considering the communication overhead for the AUV flow of 9 hops per iteration and an approximated message size of 94 bytes, we can estimate the bandwidth for the best path routing strategy($(94*8*9)/(0.0175)$=386.7 Kbps).

The factors that can contribute for the communication overhead are:

- Size of the message;

- Interprocessor distance in number of hops;

- Transputer link communication channel bandwidth(10 to 20 Mbps for the INMOS T800);

- Frequency of the main AUV flow loop iteration.($1/0.37$=2.70 Hz);

- Task switching(executed every 1ms[ALSYS 90] with an overhead of about 20 microseconds per switch);

- Circular buffer scan delay(queue delay);

- Overhead due the comparisons used by the case statement in function WHERE_IS of the package COMMON;

- Overhead due the comparisons used by the case statements used by procedures SEND_IT on each processor;

The size of the message and the interprocessor distance in number of hops are the factors that have a major influence on the communication overhead.

E.   CHANGES REQUIRED TO MEET NEW PROJECT SPECIFICATIONS

Unfortunately, the scheme supported by the Alsys ADA Compilation System is very inflexible to changes in the network topology. Therefore, at least the following modifications are required to adapt the communication layer software to different topologies:

- Existent OCCAM harnesses must be modified to reflect the new network topology so as to include the allocation of extra communication links to extra processors.

- Two extra OCCAM harnesses and an extra ADA program have to be produced for each additional transputer node to be included in the network.

- The package COMMON must be modified in order to update the routing tables.

- The initialization messages that are used by programs EARTH.ADA, MARS.ADA, PLUTO.ADA, SATURN.ADA and VENUS.ADA to

set the routing strategy and the period of repetition of the simulation flow must be modified.

In addition, in order to simulate different applications the following changes must be implemented:

- The types TASKS and ENTRYS in package COMMON must be redefined.

- The SEND_IT procedures in programs MARS.ADA, PLUTO.ADA, SATURN.ADA and VENUS.ADA must be rewritten to reflect the new task entry points.

- The body and specification of the tasks in programs MARS.ADA, PLUTO.ADA, SATURN.ADA and VENUS.ADA must be modified in order to represent the new allocation.

III. TASK ALLOCATION

A. PROBLEM DEFINITION

The problem of task allocation can be defined formally by using graph theory. The application flow can be represented by a graph $G_t = (V_t, E_t)$, where V_t is a set of vertices representing tasks $V_t = \{t_1, t_2, \ldots, t_n\}$ and E_t is a set of directed edges representing the intertask dependency. Data structures can be used in the vertices to hold task information such as execution times, deadlines and repetition rates, and in edges to hold communication costs such as the size of the message. Figure 5 shows an example of graph representation of the AUV simulation flow.

The network topology can also be represented by a graph $G_p = (V_p, E_p)$, where V_p is a set of vertices representing processors $V_p = \{p_1, p_2, \ldots, p_m\}$ and E_p is a set of links representing communication channels. Data structures can be used in the vertices to hold information about the processor such as memory capacity and performance in MIPS, and the edges can hold information about the communication channels such as bandwidth in Mbps. Figure 2 shows an example of graph representation of the transputer network used in this thesis.

The problem of task allocation usually can be seen as a many-to-one mapping, as shown in Figure 7. The problem can be

stated as how to generate an efficient mapping from N tasks to
M processors to minimize the intertask communication costs, to
balance the load utilization on the processors, to meet
individual task deadlines or to minimize the total response
time of the simulation flow. Task allocation is a
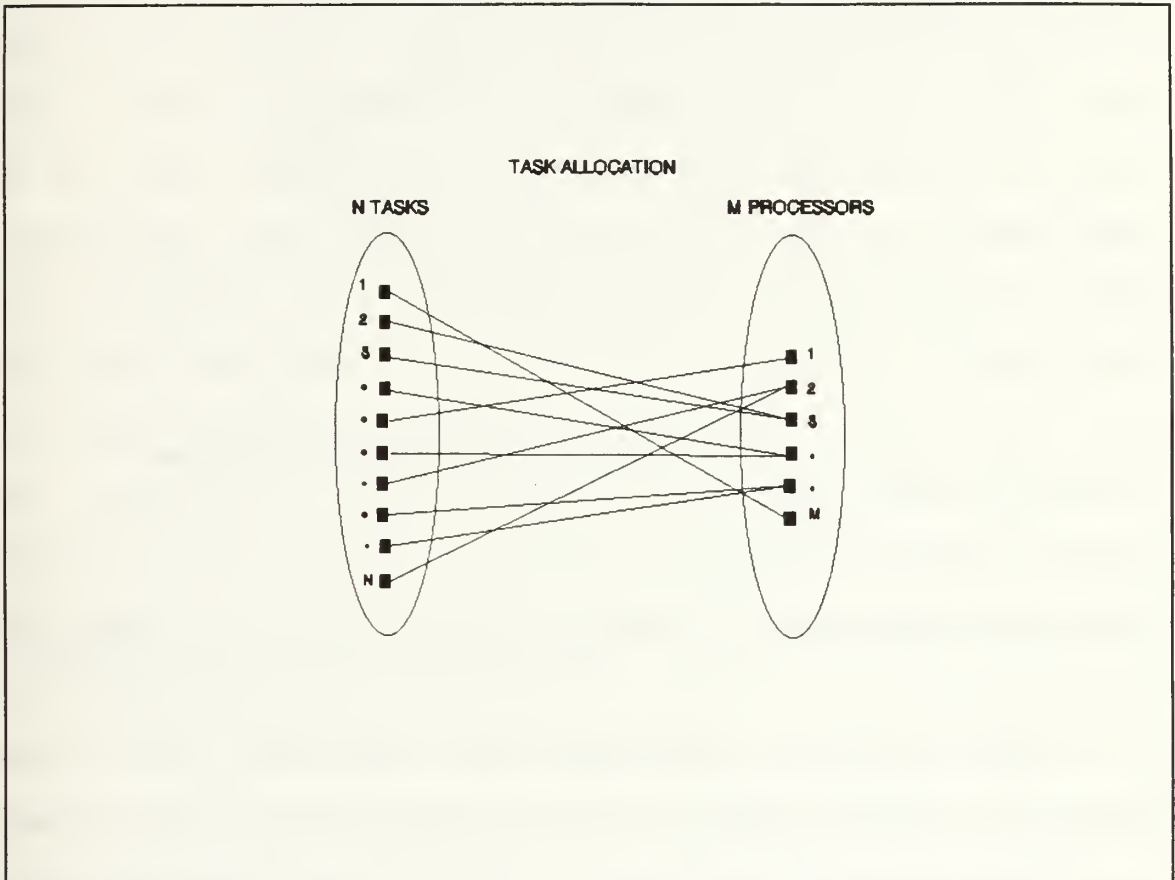
Figure 7: Task allocation as a mapping Problem

combinatorial problem whose search space S(number of possible
solutions) for a problem with N tasks and M processors can be
calculated by the following equation:

$$S = M^N \tag{1}$$

An objective function must be defined to search for a solution in the space S. This function must establish the main goal of the problem because some of the potential goals of the optimization may be contradictory. For instance, the best solution to minimize interprocess communication costs may lead to a load unbalanced system. For this thesis, the objective function tries to minimize the response time for the entire task flow and the number of processors used. Some simplifications are assumed to this problem such as no requirements to meet individual task deadlines, application task flow pre-sorted in topological order, only periodic tasks considered in the application flow, all tasks executed with the same rate of repetition ,and deterministic execution times(the worst case task execution time is employed). The latter simplification is unrealistic because task execution times can have large variances, but at least it makes the static allocation of tasks to processors possible[QUINN 87].

Even with the simplifications mentioned above, this problem is classified as NP-hard, meaning that it is unlikely that a polynomial-time algorithm could always find an optimal allocation given an arbitrary task flow graph. As an example, consider the case where the number of processors is equal to 4 and the number of tasks is equal to 16. The search space as calculated by Equation (1) results 4,294,967,296 possible solutions. If an exhaustive search were used for this problem with an assumed cost of 0.1msec per search the complete search
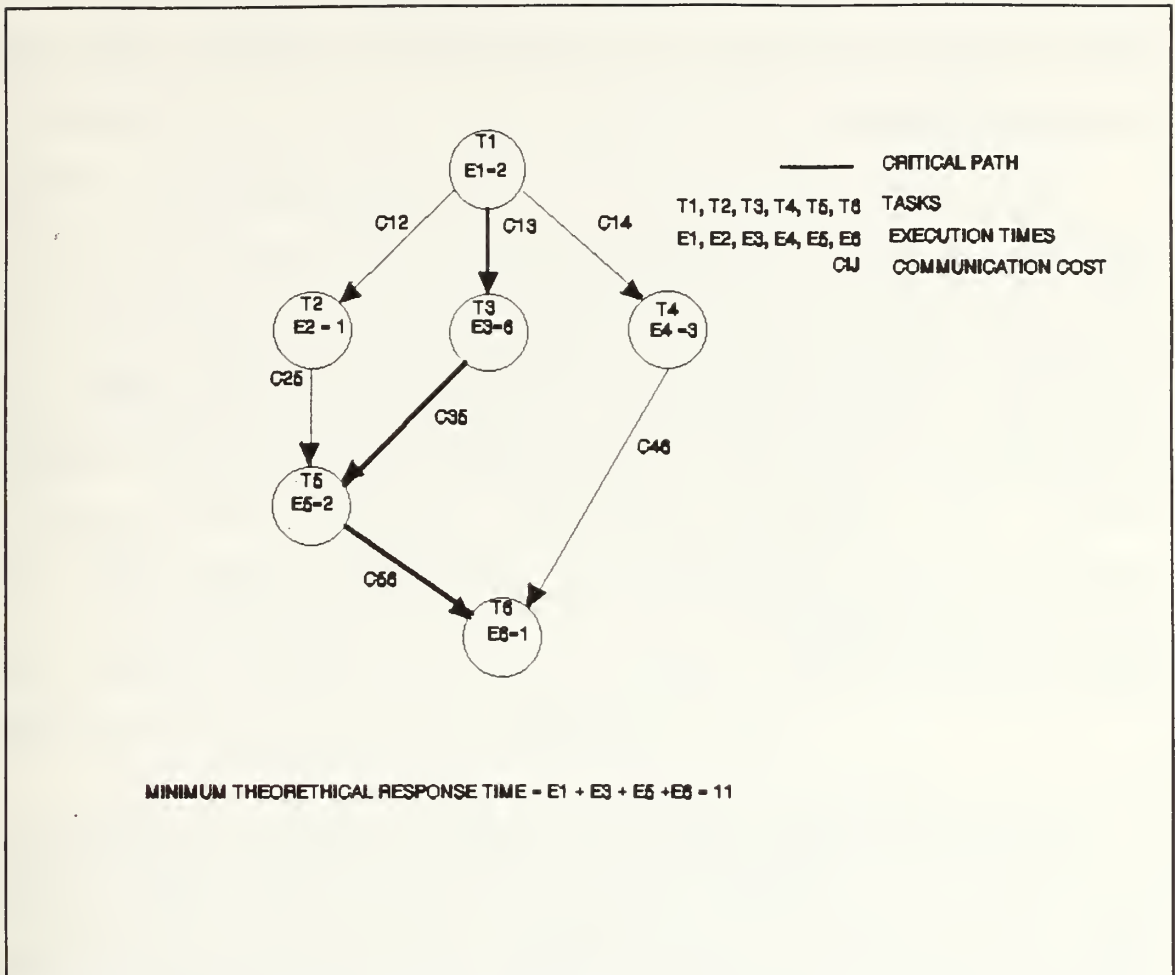
Figure 8: Minimum response time

would be completed in approximately five days. Thus, even small instances of the problem are hard and its complexity grows exponentially.

Therefore, heuristics are required to reduce the search space leading to near optimal solutions in polynomial time.

B.   DESCRIPTION OF THE HEURISTIC

A constructive assignment of tasks to processors is employed to obtain a near to optimal solution for the minimum

response time of an arbitrary application task flow with a minimum number of processors. Different aspects are considered in this scheme.

First, we notice that the best solution that can be achieved is lower bounded by the sum of the execution times of the tasks that are in the critical path, as illustrated in Figure 8. This feature is independent of the number of processors used in the network. Equation (2) formalizes how to compute the minimum theoretical response time R_{tmin} as a function of the task execution times E_i for every task i on the critical path.

$$R_{t_{min}} = \sum_i E_i \qquad (2)$$

The real minimum task flow response time for a network of processors is increased mainly by two sources that are the delay caused by task dependencies and the task communication overhead.

Task dependencies can usually be minimized by using more processors on the network to match the parallelism on the network with the parallelism on the task flow. This solution is not always satisfactory because increasing the number of processors may lead to an underutilized system. Communication overhead can be minimized by allocating tasks with high communication costs on the same processor and by having an efficient communication system.

The minimum response time R_t can be calculated as the sum of its minimum theoretical R_{tmin} the overhead due task dependencies O_d and the overhead due the communication O_c as stated in Equation (3).

$$R_t = R_{t_{min}} + O_d + O_c \qquad\qquad (3)$$

The overhead due task dependencies O_d is independent of the communication system. It happens because a child task cannot start execution before all its parent tasks finish execution.

The overhead due the communication system O_c is a function of several factors such as message size, interprocessor distances, bandwidth and queueing delay.

There is no possible alternative to decrease R_{tmin} for a given task flow because this is an intrinsic feature of the task flow. It can only be changed by modifying the task flow.

Two other parameters that are frequently used to evaluate the performance of parallel computers are the speed-up and the throughput. The speed-up is defined as the ratio of the total execution time of the task flow on an uniprocessor computer to the execution time on the parallel computer. The throughput is a measure of the number of computations of the total task flow in a given time interval.

The total execution time of the application task flow on a uniprocessor computer T_{uni} can be calculated as the sum of

the task execution times E_j for every task j on the application task flow, as stated in Equation (4).

$$T_{uni} = \sum_j E_j \qquad (4)$$

Assuming that the application task flow cannot be pipelined due to task dependencies. Then, the speed-up Sp can be calculated by Equation (5).

$$S_p = \frac{T_{uni}}{R_t} \qquad (5)$$

As our heuristic has also a goal of minimizing the number of used processors, we can estimate what is the minimum number of processors to be tried in the search for a solution. We know that there is upper bound for the speed-up that can be calculated as in Equation (6) but there is also an upper-bound for this parameter imposed by the number of processors employed in the network M as shown in Equation (7).

$$S_p \le \frac{T_{uni}}{R_{t_{min}}} \qquad (6)$$

$$S_p \le M \qquad (7)$$

From (6) and (7) we can derive a lower bound to the number of processors to be used as shown in Equation (8).

$$M \geq \frac{T_{uni}}{R_{t_{min}}} \qquad \textbf{(8)}$$

An important rule to be applied in the heuristic is to group together all tasks that compose the critical path and allocate this cluster of tasks to the same processor p_1. Then, we allocate the remaining tasks in processors p_2 to p_m. From Equation (1) we can notice that this rule reduces the search space. For instance, if there are N_1 tasks on the critical path out of the total of N tasks on the task flow the new search space S_{new} can now be calculated by Equation (9).

$$S_{new} = (N - N_1)^{(M-1)} \qquad \textbf{(9)}$$

A second rule should minimize the total overhead imposed to the response time O_{total} described in Equation (10).

$$O_{total} = O_d + O_c \qquad \textbf{(10)}$$

This rule is based on a pair-wise examination of communication tasks[RAMAMRITHAM 89]. Every pair of tasks must be examined according to one heuristic function. This heuristic function must reflect the utility of putting one particular pair of tasks together on the same processor or on separate processors. This function depends on C_{ij} that is the communication cost between two tasks T_i and T_j and on the execution times of these tasks, E_i and E_j respectively.

Equation (11) shows how the function H_{ij} balances the influence of these factors by tunable constants K_1 and K_2.

$$H_{ij} = K_1 \cdot C_{ij} + \frac{K_2}{E_i + E_j} \qquad \textbf{(11)}$$

The higher the value of this heuristic function, the higher is the utility in putting this pair of tasks on the

```
Construct the task flow graph;
Determine the critical path and the minimum
    number of processors to be used;
Allocate cluster of tasks in the critical path to
    processor P1;
For every pair of tasks {i,j}
    Calculate heuristic function h(i,j);
Sort the values of h(i,j) in ascending order;
/* Allocate tasks to processors P2 to Pm */
While not all tasks are allocated loop
    Pick not allocated  task pair {i,j} with the lowest h(i,j);
    Allocate Ti and Tj to different processors;
    Pick not allocated task pair {k,l} with the highest h(k,l);
    Allocate Tk and Tl to the same processor;
end loop;
```

Figure 9: Pseudocode for the constructive assignment

same processor. On the other hand, its lower value indicates more advantage in putting them on separate processors. The communication cost C_{ij} will be assumed zero when there is no precedence relationship between tasks T_i and T_j. Therefore, tasks with no precedence relationship tend to have a lower value of H_{ij} reflecting a tendency to go on separate

processors. This is a good consequence because tasks with no precedence relationships are good candidates for parallel execution.

In addition, the term that is inversely proportional to the sum of the execution times E_i and E_j takes into consideration the following aspects:

- The lower the sum of E_i and E_j the lower is the advantage obtained by running them in parallel even if they do not present a precedence relationship.

- On the other hand, the higher the sum of E_i and E_j, the better it is to have them separated because it balances the processor utilization. Also, tasks i and j are more likely to have some overlap on their execution times.

The basic scheme employed in the task allocation is summarized in Figure 9.

The sorted vector h(i,j) is searched from top-down and from bottom-up to allocate tasks that are not on the critical path. Tasks with lower h(i,j) values have priority to be allocated in separate processors while tasks with higher h(i,j) values have priority to be allocated on the same processor. The least used processor criterion is used to determine which processors to employ for the pair allocation to balance the load on the network.

Table XII is an example of calculation of the heuristic vector h(i,j) for the task flow of Figure 8 assuming $K_1 = 0.5$, $K_2 = 0.5$, and all communication costs equal to 1.0.

Considering the example of Figure 8, we obtain T_{uni} = 15 and R_{tmin} = 11. By using Equation (8), we get that M should be greater or equal 15/11=1.36. Therefore, two or more processors are required to achieve the best speed-up.

From Figure 8, we know that the critical path is composed of tasks T_1, T_3, T_5 and T_6. Thus, this set of tasks is allocated to processor p_1.

This example is too simple to fully demonstrate how the constructive assignment of tasks works but at least we can follow its search mechanism. Initially, the heuristic examines the pair of tasks $\{T_3, T_4\}$; we should put T_3 and T_4 in separate processors and T_3 is already allocated to p_1. So, we allocate T_4 to processor p_2. Then, the heuristic examines the pair of tasks $\{T_2, T_5\}$; we should put T_2 and T_5 on the same processor and, T_5 is already allocated to p_1, so we skip this pair. Finally, the heuristic examines the pair of tasks $\{T_2, T_3\}$. We should put T_2 and T_3 in separate processors. T_3 is already allocated to p_1, so we allocate T_2 to processor p_2 and we are done because all tasks are allocated. If the allocation were not complete then the following sequence of pairs would be searched($\{5,6\}, \{3,6\}, \{1,2\}, \{4,5\}, ...$).

Table XII: HEURISTIC FUNCTION

i	j	E_i	E_i	C_{ij}	H_{ij}
3	4	6.0	●.0	6.0	0.056
2	3	1.0	6.0	0.0	0.071
3	6	6.0	1.0	6.0	0.071
4	5	3.0	2.0	6.0	0.100
2	4	1.0	3.0	0.0	0.125
1	5	2.0	2.0	0.0	0.125
1	6	2.0	1.0	6.0	0.167
2	6	1.0	1.0	6.0	0.250
3	5	6.0	2.0	1.0	0.562
1	3	2.0	●.0	1.0	0.562
1	4	2.0	3.0	1.0	0.600
4	6	●.0	1.0	1.0	0.625
1	2	2.0	1.0	1.0	0.667
5	6	2.0	1.0	1.0	0.667
2	5	1.0	2.0	1.0	0.66

C. LIMITATIONS OF THE CONSTRUCTIVE ASSIGNMENT

The constructive assignment heuristic presented in the last section has some limitations that recommend the use of an iterative improvement to increase the quality of the task allocation.

The main limitation is that the heuristic does not take into account the interprocessor distances because these distances are only known after the allocation is completed.

49

The communication costs only consider the relative sizes of the task synchronization messages.

In addition, the heuristic function does not provide a completely ordered set of task pairs. We can notice in table XII that there are many pairs of tasks with the same value of heuristic function h(i,j).

An improvement of the task allocation can be obtained by applying the approach described in Figure 10.

```
For every processor node Pi on the current allocation loop
    Calculate the schedule of execution using task
        precedences and get the latest execution times;
    Calculate the communication overhead using
        communication costs and the interprocessor
        distances;
end loop;
Calculate the estimated maximum response time for the
    current allocation(RTcurrent);
For K in 1..NUM loop /* Repeat NUM times */
    Change the current allocation using a pair-wise
        interchange of tasks and get a new allocation;
    For every processor node Pi on the new allocation loop
        Calculate the schedule of execution using task
            precedences and get the latest execution times;
        Calculate the communication overhead using
            communication costs and the interprocessor
            distances;
    end loop;
    Calculate the estimated maximum response time for the
        new allocation(RTnew);
    if RTnew < RTcurrent then
        Current allocation := New allocation;
    end if;
end loop;
```

Figure 10: Pseudocode for task allocation improvement

Equation (12) describes how to calculate the communications overhead on processor $1(O_{cl})$ as a function of the edge communication costs C_{ij} and the interprocess

50

communication distances D_{ij} for every task i in processor l and any task j in other processors.

$$O_{c_l} = \sum_i \sum_j C_{ij} \cdot D_{ij} \qquad\qquad \textbf{(12)}$$

The pair-wise exchange of tasks first divides the set of tasks in two sub-sets of tasks by using a random number

Table XIII: PACKAGES AND THEIR FUNCTIONS USED IN ALLOCATION

PACKAGE	STATUS	FILE	FUNCTION
RANDOM	Library	N/A	Generate random numbers
TEXT_IO	Library	N/A	Input/ Output
QUEUES_2	User Defined	QUEUES2.ADA	Generic Queue abstract data type
DISCRETE_SET	User Defined	DISET.ADA	Generic Discrete Set abstract data type
SORT_ADT	User Defined	SORT.ADA	Generic Sort routines
GRAPH2_ADT	User Defined	GRAPH2.ADA	Generic Directed Graph abstract data type

51

generator. Then, it exchanges every item in the first sub-set with the corresponding item in the second sub-set. This exchange should only be performed if neither task is on the critical path.

D. ALLOCATION IMPLEMENTATION

A general overview of the heuristic described in the previous section is illustrated in Figure 10. The complete scheme is implemented by the main procedure STATICAL in file
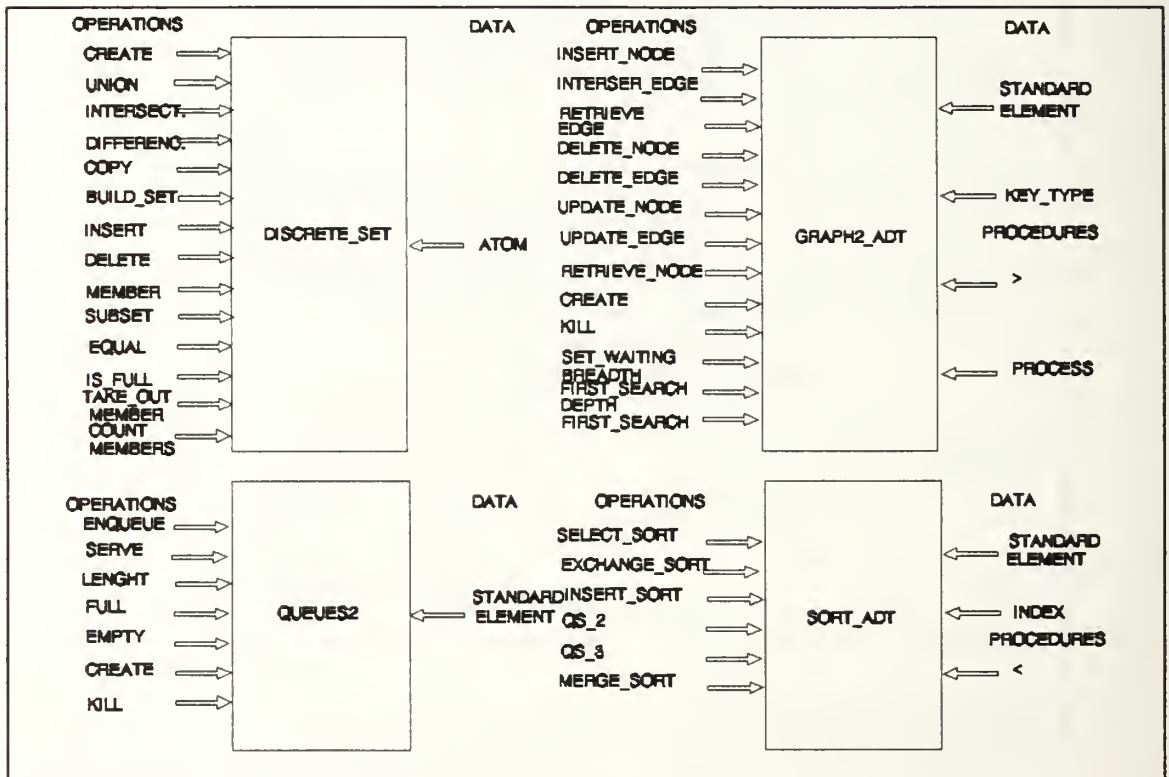
Figure 11: User defined packages

STATICAL.ADA. Table XIII shows the packages that are used by this program. The packages QUEUES2.ADA, DISET.ADA, SORT.ADA

and GRAPH2.ADA are enclosed in Appendix C. The separately compiled procedures CONSTRUCT_TASK_FLOW, CALC_HEURISTIC, ALLOCATE, SCHEDULE and IMPROVE are called by the main program STATICAL.

The files STATICAL.ADA, CTFLOW.ADA, CALHEU.ADA, ALLOC.ADA, SCHED.ADA and IMPROVE.ADA are enclosed in Appendix D.

Figure 11 shows the basic operations offered by each one of the abstract data types as defined in the generic packages QUEUES2, DISCRETE_SET, GRAPH2_ADT and SORT_ADT. These packages are ADA versions of the MODULA-2 implementations described in [STUBBS 87].

Package DISCRETE_SET implements the SET abstract data type that is very useful in task allocation. For instance, each node allocation is a SET of tasks that run on a particular processor node.

Package GRAPH2_ADT implements the directed-graph(DGRAPH) abstract data type that is used to represent the application task flow.

Package QUEUES2 implements a FIFO queue that is used as an auxiliary data structure for the calculation of the critical path of the task flow.

Package SORT_ADT implements several schemes of SORT. A quick sort algorithm(QS_3) is employed to sort the heuristic vector $h(i,j)$.

Package RANDOM is employed to generate the two clusters of tasks that are used in the iterative pair-wise exchange task

53
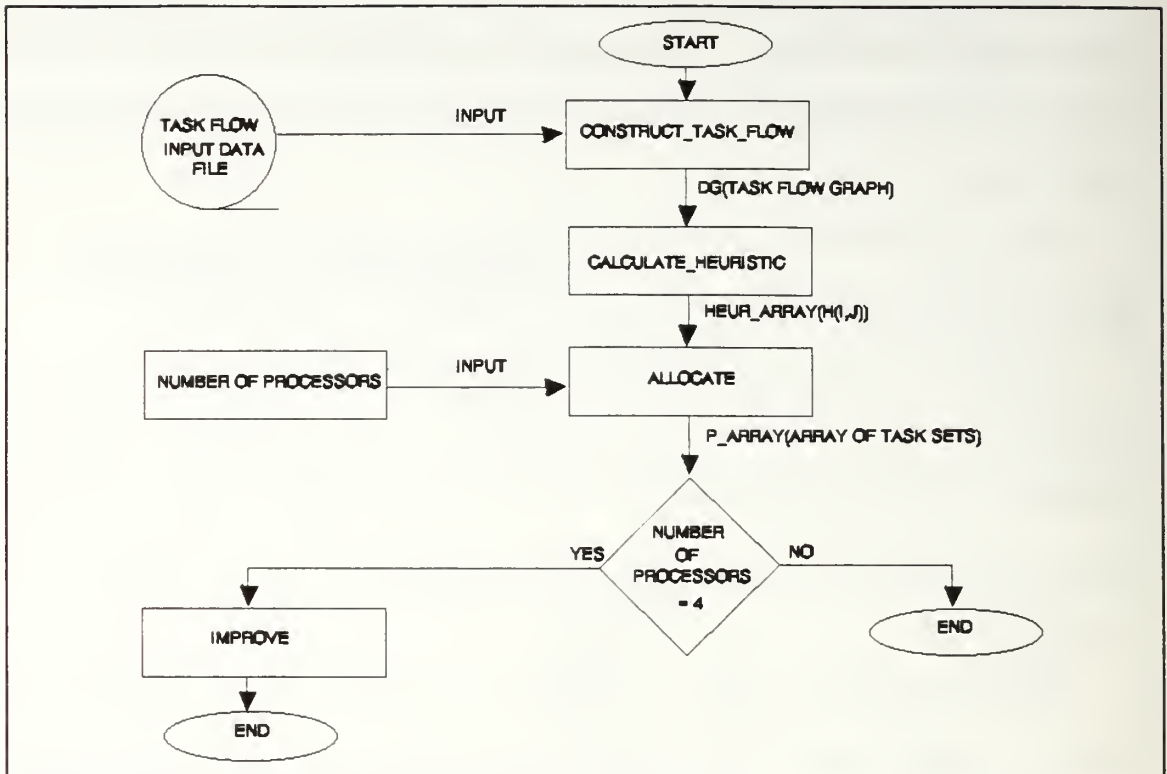
Figure 12: STATICAL data flow diagram

for the allocation improvement.

Figure 12 shows the data flow diagram of the main program STATICAL. This program has the following basic structure:

- Initially, it calls the procedure CONSTRUCT_TASK_FLOW. This procedure reads the data file describing the application task flow, and then generates a directed task flow graph using the operators INSERT_NODE and INSERT_EDGE of the package GRAPH2_ADT. In addition, it counts the number of tasks and calculates the critical path for the task flow.

- It calls the procedure CALC_HEURISTIC that calculates the heuristic function $h(i,j)$ for every pair of tasks $\{i,j\}$ and then sorts the vector $h(i,j)$ using a quicksort algorithm.

54

- It calls the procedure ALLOCATE that queries the user to input the number of processors to be used and then allocate tasks to the processors following the constructive assignment.

- It calls the procedure SCHEDULE that calculates the schedule of execution on each processor. It employs an algorithm that schedules tasks with lower identification numbers first. It is assumed that the input task flow had already been submitted to a topological sorting.

- Finally, if the number of processors is equal to 4 it will call the procedure IMPROVE that enhances the task allocation using the iterative pair-wise interchange of tasks.

FILE : FLOW.DAT

```
EXAMPLE - FLOW WITH 6 TASKS
NODE  1   2.0
NODE  2   1.0
EDGE  1    2      1.0
NODE  3   6.0
EDGE  1    3      1.0
NODE  4   3.0
EDGE  1    4      1.0
NODE  5   2.0
EDGE  2    5      1.0
EDGE  3    5      1.0
NODE  6   1.0
EDGE  4    6      1.0
EDGE  5    6      1.0
EDGE  6    1      1.0
```

Figure 13: Task flow input data file

55

The task flow input data file must be specified using the commands NODE and EDGE. The command NODE must be followed by
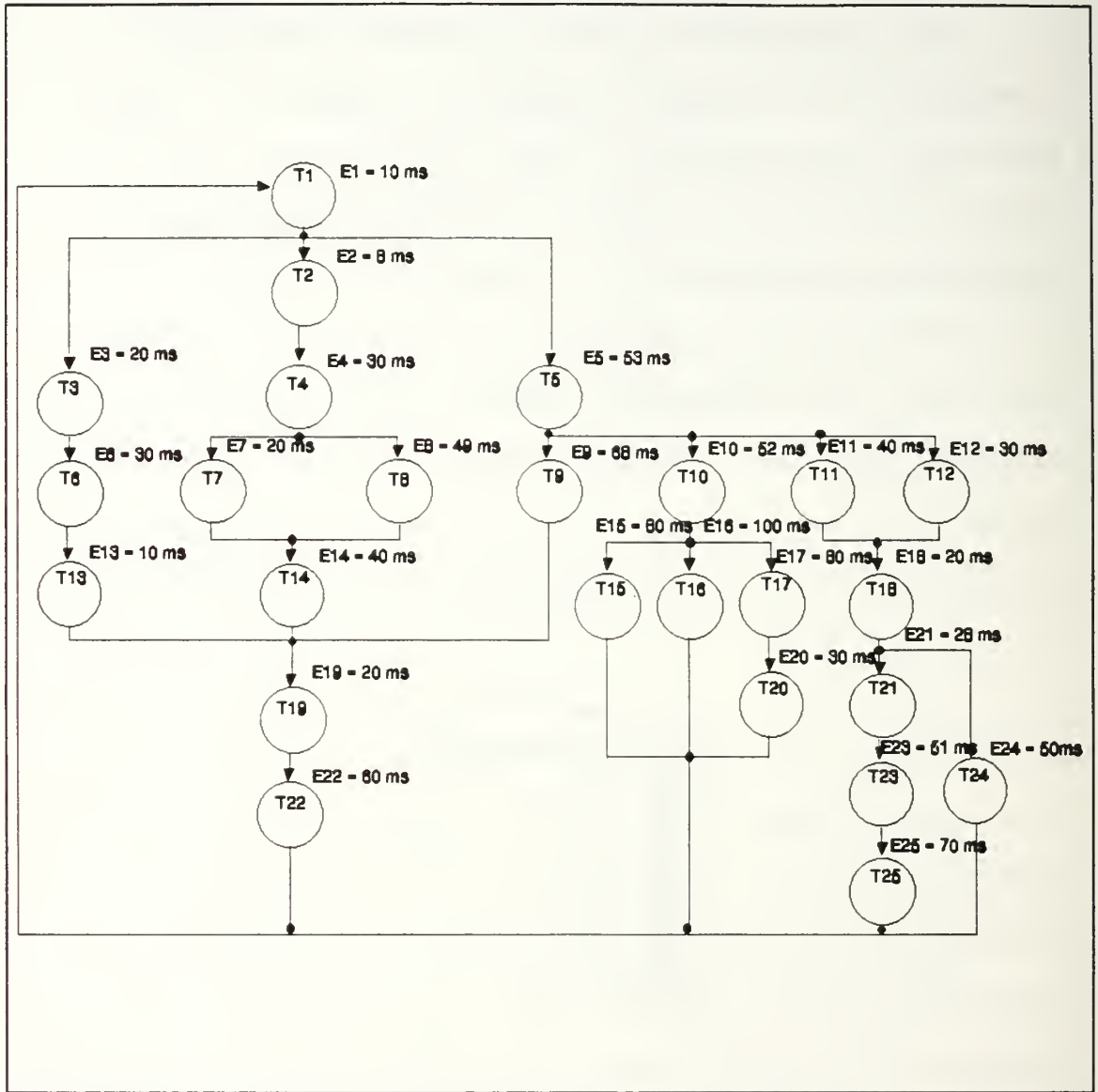
Figure 14: 25-Task simulation flow

the task identification number of type INTEGER and by the task execution time of type FLOAT. The command EDGE must be followed by the source and destination task identifiers both of type INTEGER and by the communication cost of type FLOAT.

56

The first line of the file is not used because it is reserved for comments. Figure 13 shows one example of task flow input data file for the flow described in Figure 8 with communication costs equal to 1.0. We can notice that an extra edge is placed between task T_6 and task T_1 to characterize the assumption that this flow cannot be pipelined.

E. PERFORMANCE RESULTS

1. APPLICATION FLOW WITH 25 TASKS

The 25-task example of Figure 14 was simulated using the transputer network of Figure 2 to verify the robustness of the communication layer in relation to deadlocks with a more complex flow than the AUV flow and to evaluate the constructive assignment heuristic. This flow is a modified version of the Air-Defense flow described in [TSUCHIYA 82]. The allocation for this flow considering all communication costs equal to 1.0, $K_1 = 0.5$ and $K_2 = 0.5$ is shown in Figure 16.

Table XIV shows the results obtained for this flow with different routing strategies:

Table XIV: RESULTS FOR THE FLOW OF FIGURE 14

Routing Strategy	Counter clock- wise ring	Clock- wise ring	Multi- Path	Best Path
Execution time(ms)	380.62	379.87	378.04	377.43

By running the program STATICAL we can get that T_{uni} = 1049 ms and that the longest path is composed of tasks T_1, T_2, T_5, T_{11}, T_{18}, T_{21}, T_{23} and T_{25} with R_{tmin} = 280 ms and T_{uni}/R_{tmin} = 3.7464. Thus, by Equation (8) we should use at least 4 processors. This program also gives Od = 45ms and Rtmin+Od =
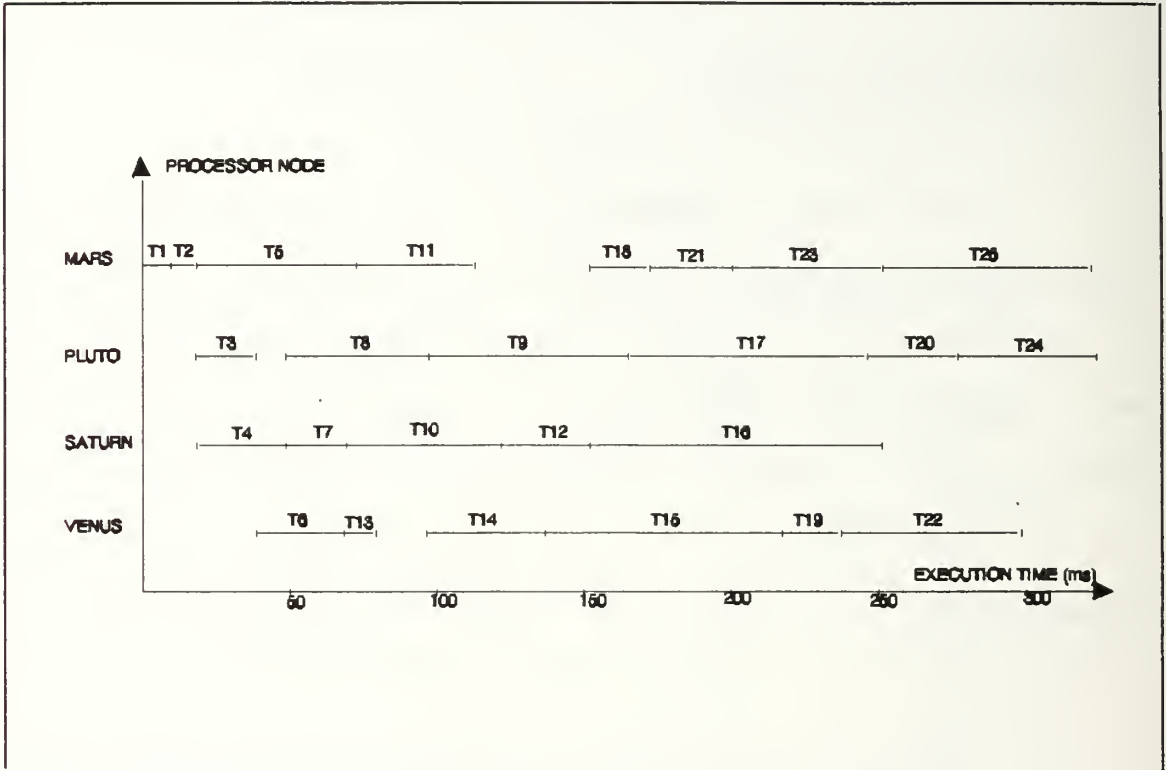
Figure 15: Task allocation for the flow of Figure 14

325.0ms when using 4 processors. Therefore, we should use more than 4 processors to drive the dependency overhead to zero. The communication overhead can be obtained by using the data in Table XIV that gives Oc = 377.43 - 325.0 = 52.43 ms. The communication efficiency can be calculated by Equation (13) that gives Ec = 86.10 %.

$$E_c = \frac{R_{t_{min}} + O_d}{R_{t_{min}} + O_d + O_c} \qquad (13)$$

The processor utilization Up can be calculated by Equation (14). Table XV shows the utilization for every node processor. From this data, we can notice that the allocation scheme resulted in a load balanced system.

$$U_p = \frac{\sum\limits_{i \in p} E_i}{R_t} \qquad (14)$$

Table XV: UTILIZATION FOR THE FLOW OF FIGURE 14

NODE	MARS	PLUTO	SATURN	VENUS
Up(%)	74.18	78.69	61.47	63.59

Since we do not have automatic tools for changing the allocation on the network of transputers we will investigate how to improve the constructive assignment and how the allocation changes to reflect different communication costs using a simpler flow with 16 tasks. This simplification is due the amount of work required to change the allocation with a larger flow without automatic tools and does not imply loss of generality.

2. APPLICATION FLOW WITH 16 TASKS

Figure 16 shows the task application flow that is used in this sub-section. It is an arbitrary task flow with several features commonly found in real applications such as task dependencies and threads of parallelism with fork/joint structures.

a. CONSTRUCTIVE ASSIGNMENT WITH EQUAL COMMUNICATION COSTS
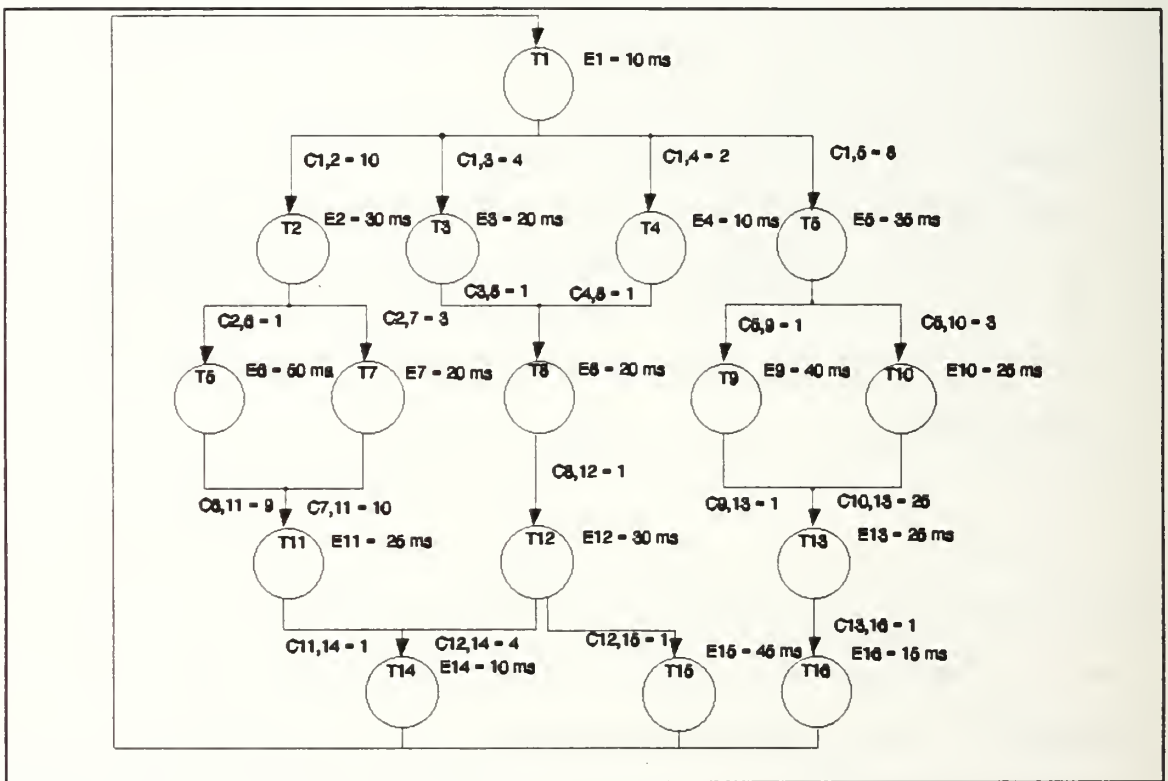
Figure 16: 16-Task simulation flow

Figure 17 shows the allocation generated by the constructive assignment heuristic considering all communication costs equal to 1.0, $K_1 = 0.5$ and $K_2 = 0.5$.
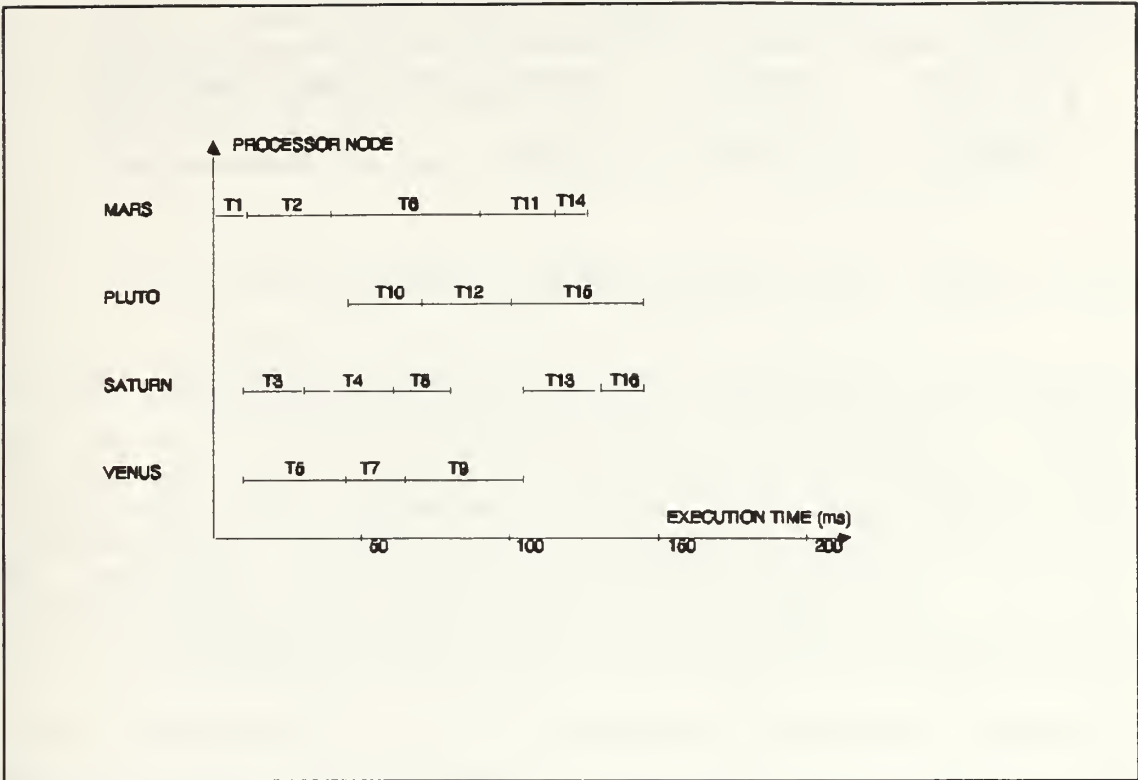
Figure 17: Constructive assignment with equal communication costs

Table XVI presents the results obtained with this task allocation on the transputer network of Figure 1.

Table XVI: RESULTS FOR THE CONSTRUCTIVE ASSIGNMENT WITH EQUAL COMMUNICATION COSTS

Routing strategy	Counter clock-wise ring	Clock-wise ring	Multi-path	Best Path
Execution time(ms)	170.75	163.91	163.70	163.97

This flow has T_{uni} = 410.0 ms and its longest path is composed by tasks T_1, T_2, T_6, T_{11} and T_{14} with R_{tmin} = 125 ms

61

Table XVII: UTILIZATION FOR THE CONSTRUCTIVE ASSIGNMENT ALLOCATION WITH EQUAL COMMUNICATION COSTS

NODE	MARS	PLUTO	SATURN	VENUS
Up(%)	76.23	60.99	54.99	57.94

resulting in T_{uni}/R_{tmin} = 3.28. With the allocation shown in figure 17, we get O_d = 20 ms and $R_{tmin}+O_d$ = 145 ms. The communication overhead can be obtained by using the data of Table XVI resulting in O_c = 163.97-145.0 = 18.97 ms. The communication efficiency, as calculated by Equation (13), is
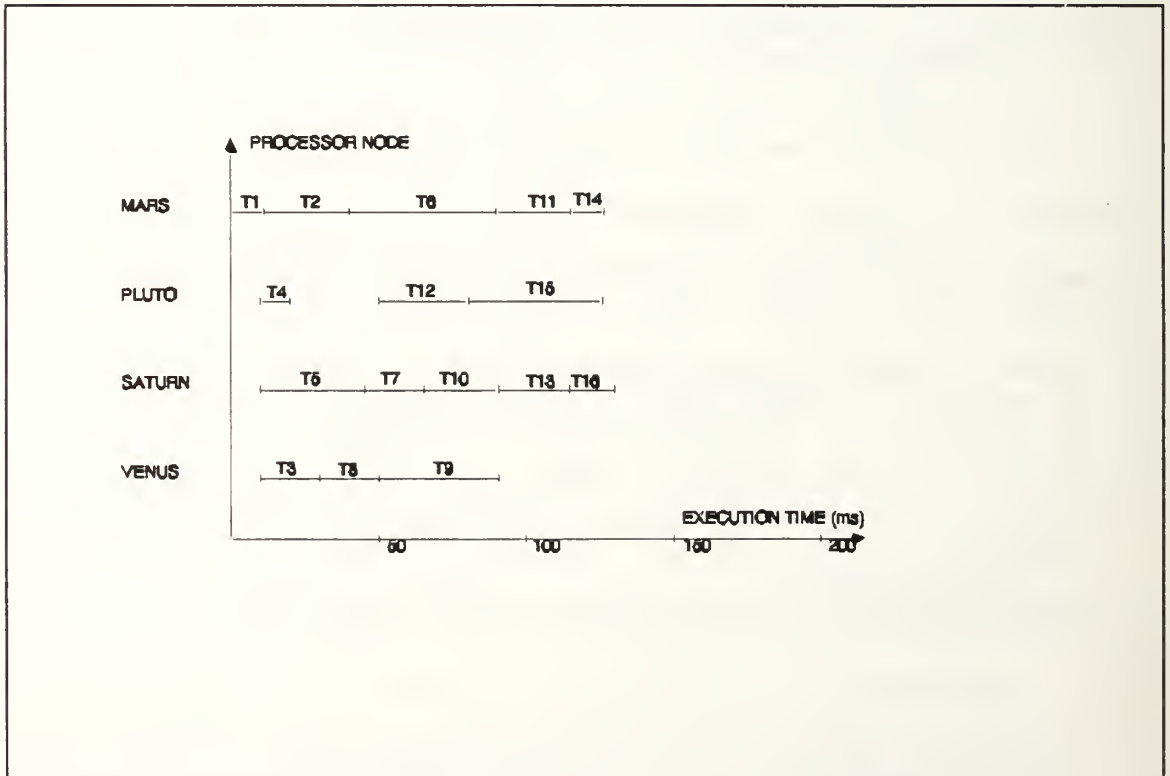
Figure 18: Improved task allocation with equal communication costs

$E_c = 88.43$ %. The resulting processor utilization on each node processor is shown on Table XVII.

b. *IMPROVING THE ALLOCATION WITH EQUAL COMMUNICATION COSTS*

Figure 18 shows the task allocation generated by improving the original allocation considering equal communication costs and by using 100 iterations of the iterative pair-wise tasks interchange algorithm.

Table XVIII presents the results obtained with this task flow simulation on the transputer network of Figure 2.

Table XVIII: RESULTS WITH THE IMPROVED ALLOCATION USING EQUAL COMMUNICATION COSTS

Routing strategy	Counter clock- wise ring	Clock- wise ring	Multi- path	Best Path
Execution time(ms)	145.47	142.05	141.99	142.23

Table XIX: UTILIZATION FOR THE IMPROVED ALLOCATION WITH EQUAL COMMUNICATION COSTS

NODE	MARS	PLUTO	SATURN	VENUS
Up(%)	87.88	87.88	91.4	63.27

With this allocation, we get O_d = 5 ms and $R_{tmin}+O_d$ = 130ms. The communications overhead can be obtained by using the data of Table XVIII resulting O_c = 142.23 - 130 = 12.23 ms. The communication efficiency, as calculated by Equation (14), is E_c = 91.4 %. The resulting processor utilization on each node processor is shown in Table XIX.

c. ADDING DIFFERENT COMMUNICATION COSTS TO THE IMPROVED ALLOCATION

In this part, we simulate the allocation presented in Figure 18 but using the different communication costs shown in Figure 16. These costs are simulated by sending task synchronization messages more than once. For instance, Figure 16 specifies C_{12}= 10, therefore 10 messages are sent from task T_1 to task T_2. Table XX shows the results obtained with this simulation.

Table XX: RESULTS WITH DIFFERENT COMMUNICATION COSTS IN THE FLOW OF FIGURE 16

Routing strategy	Counter clock- wise ring	Clock- wise ring	Multi- path	Best Path
Execution time(ms)	216.28	211.56	208.73	208.97

The communications overhead is obtained by using the data on table XX resulting O_c = 208.97 - 130.0 = 78.97 ms. The communications efficiency, as calculated by Equation (13), is E_c = 62.21 %. The resulted processor utilization on each

Table XXI: UTILIZATION WITH DIFFERENT COMMUNICATION COSTS

NODE	MARS	PLUTO	SATURN	VENUS
Up(%)	59.82	40.66	57.42	38.28

node processor is shown on table XXI.

The constructive assignment and the iterative improvement must reflect communication costs and interprocessor distances. We evaluate next the constructive assignment considering different communication costs and its iterative improvement.

d. IMPROVING THE ALLOCATION WITH DIFFERENT COMMUNICATION COSTS

Figure 19 shows the allocation generated by using the constructive assignment heuristic considering the task flow with the different communication costs shown in Figure 16.

In addition, Figure 20 presents the improved allocation obtained by using the pair-wise task interchange approach considering different communication costs and Table XXII shows its simulation results.

With this allocation we get O_d = 20 ms and $R_{tmin}+O_d$ = 145 ms. The communications overhead can be obtained by using the data in Table XXII resulting in O_c = 202.51 - 145 = 57.51 ms. The communications efficiency, as calculated by Equation
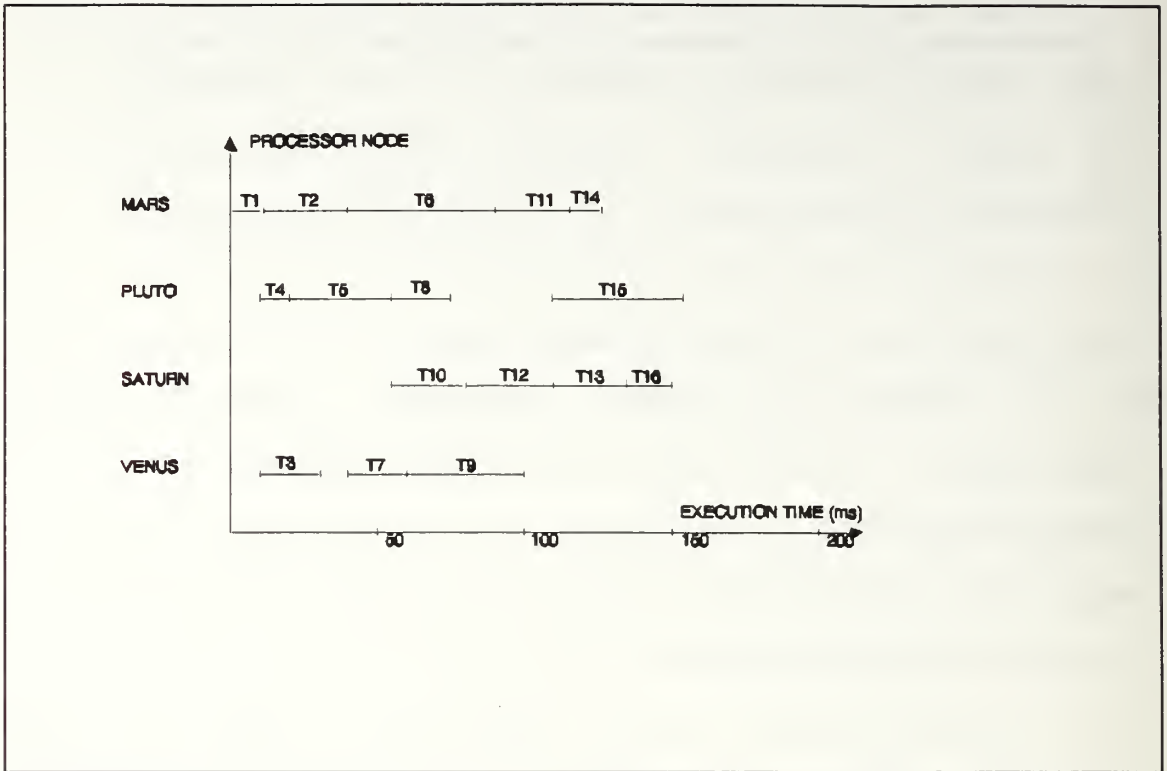
Figure 19: Constructive assignment with different communication costs

Table XXII: RESULTS FOR THE IMPROVED ALLOCATION WITH DIFFERENT COMMUNICATION COSTS

Routing strategy	Counter clock-wise ring	Clock-wise ring	Multi-path	Best Path
Execution time(ms)	206.55	212.92	209.06	202.51

(13), is E_c = 71.6%. The resulting processor utilization on each node processor is shown in Table XXIII.

We can notice that the overhead due dependencies has increased from 5ms for the allocation of Figure 18 to 20ms for

66
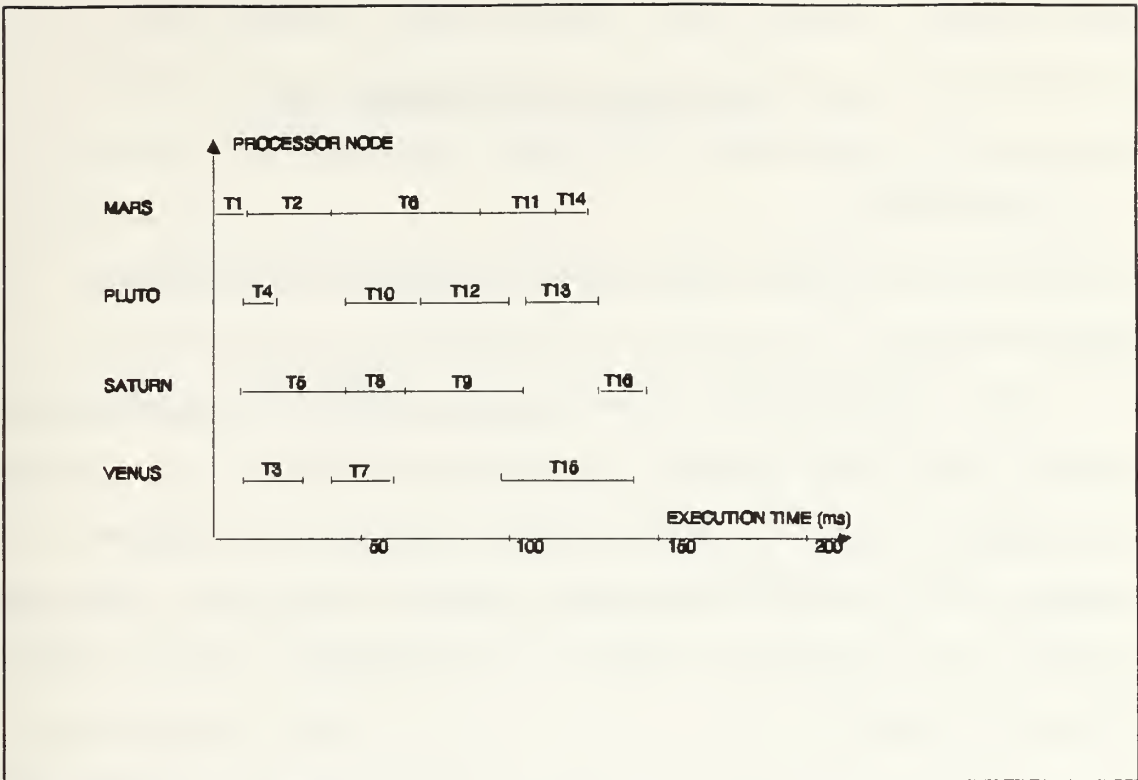
Figure 20: Improved allocation with different communication costs

Table XXIII: UTILIZATION FOR THE IMPROVED ALLOCATION WITH DIFFERENT COMMUNICATION COSTS

NODE	MARS	PLUTO	SATURN	VENUS
Up(%)	61.73	44.44	54.32	41.97

the allocation of Figure 20. However, the overall response time has been improved with the new allocation because the communication overhead has been reduced from 78.97ms to 57.51ms.

IV. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

From the communication layer description in Chapter II we conclude that:

- The communication layer package(COMLAYER) can be easily reused when encapsulated in a generic package. All other packages and procedures of the communication system will require the modifications described in section II.E when either the simulated flow/task allocation or the network topology changes.

- Different approaches for routing messages can be used with the employed network topology. The Multipath and the Best Path routing strategies have improved the communication performance in relation to the Ring routing strategy employed in the first implementation[RICHMOND 91]. In addition, these new approaches can also be used with higher order network topologies such as hypercubes or meshes.

- The communication layer has become more robust with respect to deadlocks as demonstrated by the successful simulation of one application flow with 25 tasks in sub-section III.E.1.

- The communication layer limitation of having to use only one pre-defined type for all message formats precludes its use

with more general application task flows. Despite this shortcoming, the communication layer and the language ADA are very useful to experiment task flow simulations on transputer networks.

Considering the task allocation described in chapter III and the goals of minimizing the application flow response time and the number of processors on the network, we conclude that:

- A constructive assignment heuristic can be employed to generate an initial near to optimal static task allocation taking into consideration task execution times and communication costs.

- The initial task allocation can be improved by an iterative pair-wise interchange of tasks taking into consideration the interprocess communication distances.

B. FUTURE WORK

1. ADA ON TRANSPUTER NETWORKS

The use of ADA on transputers network is not straightforward in the version of the ALSYS-ADA compiler available in the PARCDS-Laboratory. The main problem is that we need to write two OCCAM harnesses for every transputer main program. The process of writing OCCAM harnesses for every transputer node to increase the number of transputers on the network is very slow and error prone. Therefore, there is a

need for tools to generate these OCCAM harnesses automatically to make them totally transparent to the ADA user.

Another problem related to the ALSYS-ADA compiler is that it is very slow. One program for task flow simulation takes about 40 minutes to be compiled.

In addition, there are no appropriate debugging tools to support this ADA distributed development system. The programs are debugged by sending messages to the host transputer that is the only one that has access to the I/O offered by the host PC. Furthermore, when we have a deadlock situation these messages might not be delivered to the host PC and we get in a difficult situation to find out what causes the deadlock.

ALSYS-ADA is now offering a new product called ADA-MAP that promises solving these problems. A further evaluation is needed to check if this new product really solves these problems to justify additional investments on this ADA compiler.

2. TASK FLOW SIMULATOR

A task flow simulator would be a useful tool to support the study of task allocation on transputer networks. With the current approach, it is a very slow process to change the task allocation on the transputer network because, for every change, we need to rewrite parts of the programs COMMON.ADA, MARS.ADA, PLUTO.ADA, SATURN.ADA and VENUS.ADA.

A task flow simulator can be implemented by improving the program STATICAL.ADA. Besides generating the task allocation this program would also produce the programs COMMON.ADA, MARS.ADA, PLUTO.ADA, SATURN.ADA and VENUS.ADA.

For the sake of simplicity, the task flow simulator would run on the transputer network employed in this thesis. This problem is still fairly complicated because it involves text processing to generate the programs COMMON.ADA, MARS.ADA, PLUTO.ADA, SATURN.ADA and VENUS.ADA automatically from the task allocation computed in STATICAL.ADA.

3. THE INMOS T9000 TRANSPUTER

The INMOS T9000 transputer is the most recent release of the new generation of transputers. It runs at 50 Mhz, with 16-Kb cache, up to 4-gigabyte of local memory, 32 bit ALU, a 64 bit CPU, 4 bi-directional serial-communications links at 100 Mbps each, a virtual-channel processor, a programmable memory interface, two on-chip timers, four pairs of event channels for synchronizing internal processes with external events, and two control links that allow control signals to be sent between T9000s independently of the data links.

The new T9000 INMOS transputer has a packet-switched virtual communications system that takes the responsibility of routing messages from the programmer's code to significantly faster hardware reducing the communication overhead.

4. EXTENDING THE TASK ALLOCATION SCHEME

This thesis is a first step towards more elaborate schemes of task allocation on transputer networks. The following sequence of studies is suggested in order to gradually improve the current task allocation approach to face the strict requirements of the next generation of real-time parallel computers:

-Static allocation considering task placement constraints.

-Static allocation considering deadlines for the tasks that compose the application task flow.

-How the current static allocation scheme can be complemented by a dynamic allocation approach in order to consider aperiodic tasks.

-How the current static allocation scheme can be complemented by a dynamic allocation approach to consider tasks with non-deterministic execution times.

-What is the communication support and timing analysis tools required to implement a dynamic allocation scheme on a network of transputers. Investigate if the existing communication layer and the language ADA are still suitable for such a scheme.

APPENDIX A: OCCAM SOURCE CODE

A. OCCAM HARNESSES ON PROCESSOR EARTH

1. EARTHH.OCC

```
#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC earth.harness (CHAN OF SP  FromAda, ToAda,
                    CHAN OF ANY Debug,
                    CHAN OF INT Mars2Earth, Earth2Mars,
                    []INT FreeMemory)

  #IMPORT "earthh2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [3]INT in.program:
  [3]INT out.program:
  SEQ
    -- Set up vector of pointers to channels.
    in.program[0] := MOSTNEG INT    -- not used
    LOAD.INPUT.CHANNEL (in.program[1], ToAda)
    LOAD.INPUT.CHANNEL (in.program[2], Mars2Earth)
    LOAD.OUTPUT.CHANNEL (out.program[0], Debug)
    LOAD.OUTPUT.CHANNEL (out.program[1], FromAda)
    LOAD.OUTPUT.CHANNEL (out.program[2], Earth2Mars)
    -- Invoke the Ada program.
    -- Assumes the entry point name has been changed to
    -- "earth.program".
    earth.program (ws1, in.program, out.program, dummy.ws)
:
```

2. EARTHH2.OCC

```
#OPTION "AEV"

PROC earth.program ([]INT ws1, in, out, ws2)
  [1000]INT d:
  SEQ
    SKIP
:
```

73

3. MERGER.OCC

```
#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC debug.merger (CHAN OF SP FromFiler, ToFiler,
                   []CHAN OF ANY Debug,
                   CHAN OF BOOL Stop)

  #USE "hostio.lib"

  -- A debug channel merger and blocker.

  VAL max.debug IS 20:
  VAL number.of.debug IS SIZE Debug:

  INT line.index:
  [256]BYTE line.buffer:
  BYTE value, r:
  BOOL running, reset, s:
  [max.debug]BOOL mask:
  VAL BYTE line.feed IS 10 (BYTE):
  SEQ
    SEQ i = 0 FOR number.of.debug
      mask[i] := TRUE
    running := TRUE
    reset := FALSE
    line.index := 0
    WHILE running
      PRI ALT
        ALT i = 0 FOR number.of.debug
          mask[i] & Debug[i] ? value
            SEQ
              IF
                value = line.feed
                  SEQ
                    -- Send the complete line.
                    so.puts (FromFiler, ToFiler,
                          spid.stdout,
                      [line.buffer FROM 0 FOR line.index],
                    r)
                    line.index := 0
                    mask [i] := FALSE
                    reset := TRUE
                TRUE
                  SEQ
                    -- Add character to line.
                    line.buffer[line.index] := value
                    line.index := line.index + 1
        reset & SKIP
```

74

```
        SEQ
          reset := FALSE
          SEQ i = 0 FOR number.of.debug
            mask[i] := TRUE
      Stop ? s
        running := FALSE
:


    4.   MAINH.OCC


#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC main.harness (CHAN OF SP  FromFiler, ToFiler,
                   CHAN OF INT Mars2Earth, Earth2Mars,
                   []INT FreeMemory)

  #USE "hostio.lib"

  #USE "earthh.t8s"
  #USE "merger.t8s"

  [1]CHAN OF ANY Debug:
  [2]CHAN OF SP FromAda, ToAda:
  CHAN OF BOOL StopDebug, StopMultiplexor:
  SEQ

    PAR

      -- A multiplexor to combine the debug and normal
      -- output.
      so.multiplexor (FromFiler, ToFiler, FromAda, ToAda,
  StopMultiplexor)

      -- A debug channel merger.
      debug.merger (ToAda[0], FromAda[0], Debug, StopDebug)

      -- A process to invoke the earth program.
      ws IS FreeMemory:
      SEQ
        earth.harness (FromAda[1], ToAda[1], Debug[0],
      Mars2Earth, Earth2Mars, ws)
        StopDebug ! FALSE
        StopMultiplexor ! FALSE

    so.exit (FromFiler, ToFiler, sps.success)
:
```

75

5. MAIN.PGM

```
#INCLUDE "hostio.inc"
#INCLUDE "linkaddr.inc"

#USE "mainh.c8s"
#USE "marsh.c8s"
#USE "venush.c8s"
#USE "saturnh.c8s"
#USE "plutoh.c8s"

CHAN OF INT Mars2Earth, Earth2Mars, Mars2Pluto, Pluto2Mars,
Mars2Venus,Venus2Mars, Pluto2Saturn, Saturn2Pluto,
Saturn2Venus,Venus2Saturn:
CHAN OF SP FromFiler, ToFiler:

PLACED PAR

  PROCESSOR 0 T8

    PLACE FromFiler  AT link0.in:
    PLACE ToFiler    AT link0.out:
    PLACE Mars2Earth AT link2.in:
    PLACE Earth2Mars AT link2.out:

    [325000] INT ws1:
    main.harness (FromFiler, ToFiler, Mars2Earth,
  Earth2Mars, ws1)

  PROCESSOR 1 T8

    PLACE Earth2Mars AT link0.in:
    PLACE Mars2Earth AT link0.out:
    PLACE Venus2Mars AT link2.in:
    PLACE Mars2Venus AT link2.out:
    PLACE Pluto2Mars AT link3.in:
    PLACE Mars2Pluto AT link3.out:

    [280000] INT ws2:
    mars.harness (Mars2Earth, Earth2Mars, Venus2Mars,
    Mars2Venus, Mars2Pluto, Pluto2Mars, ws2)

  PROCESSOR 2 T8

    PLACE Saturn2Venus AT link2.in:
    PLACE Venus2Saturn AT link2.out:
    PLACE Mars2Venus   AT link3.in:
    PLACE Venus2Mars   AT link3.out:
```

76

```
    [280000] INT ws2:
    venus.harness (Mars2Venus, Venus2Mars, Venus2Saturn,
    Saturn2Venus, ws2)

  PROCESSOR 3 T8


    PLACE Pluto2Saturn AT link2.in:
    PLACE Saturn2Pluto AT link2.out:
    PLACE Venus2Saturn AT link3.in:
    PLACE Saturn2Venus AT link3.out:

    [280000] INT ws2:
    saturn.harness (Saturn2Venus, Venus2Saturn,
    Saturn2Pluto, Pluto2Saturn,ws2)

  PROCESSOR 4 T8

    PLACE Mars2Pluto   AT link2.in:
    PLACE Pluto2Mars   AT link2.out:
    PLACE Saturn2Pluto AT link3.in:
    PLACE Pluto2Saturn AT link3.out:

    [280000] INT ws2:
    pluto.harness (Mars2Pluto, Pluto2Mars, Saturn2Pluto,
    Pluto2Saturn, ws2)
```

B.   OCCAM HARNESSES ON PROCESSOR MARS

1.   MARSH.OCC

```
#OPTION  "AGNVW"
#INCLUDE "hostio.inc"

PROC mars.harness (CHAN OF INT Mars2Earth, Earth2Mars,
Venus2Mars, Mars2Venus,
                   Mars2Pluto, Pluto2Mars,
                   []INT FreeMemory)

  #IMPORT "marsh2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [6]INT in.program:
  [6]INT out.program:
  SEQ
    -- Set up vector of pointers to channels.
    in.program[0] := MOSTNEG INT    -- not used
    in.program[1] := MOSTNEG INT    -- standard i/o not used
```

```
      LOAD.INPUT.CHANNEL (in.program[2], Earth2Mars)
      in.program[3] := MOSTNEG INT    -- reserved for future
                                      -- use
      LOAD.INPUT.CHANNEL (in.program[4], Venus2Mars)
      LOAD.INPUT.CHANNEL (in.program[5], Pluto2Mars)
      out.program[0] := MOSTNEG INT   -- standard i/o not used
      out.program[1] := MOSTNEG INT   -- standard i/o not used
      LOAD.OUTPUT.CHANNEL (out.program[2], Mars2Earth)
      out.program[3] := MOSTNEG INT   -- reserved for future
                                      -- use
      LOAD.OUTPUT.CHANNEL( out.program[4], Mars2Venus)
      LOAD.OUTPUT.CHANNEL (out.program[5], Mars2Pluto)
      -- Invoke the Ada program.
      -- Assumes the entry point name has been changed to
      -- "mars.program".
      mars.program (ws1, in.program, out.program, dummy.ws)
  :
```

2. MARSH2.OCC

```
#OPTION "AEV"

PROC mars.program ([]INT ws1, in, out, ws2)
  [1000]INT d:
  SEQ
    SKIP
  :
```

C. OCCAM HARNESSES ON PROCESSOR PLUTO

1. PLUTOH.OCC

```
#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC pluto.harness (CHAN OF INT Mars2Pluto, Pluto2Mars,
                    Saturn2Pluto,Pluto2Saturn,
                      []INT FreeMemory)

  #IMPORT "plutoh2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [6]INT in.program:
  [6]INT out.program:
  SEQ
    -- Set up vector of pointers to channels.
    in.program[0] := MOSTNEG INT    -- not used
```

78

```
        in.program[1] := MOSTNEG INT    -- standard i/o not used
        in.program[2] := MOSTNEG INT    -- reserved for future
                                        -- use
        LOAD.INPUT.CHANNEL (in.program[3], Mars2Pluto)
        LOAD.INPUT.CHANNEL (in.program[4], Saturn2Pluto)
        in.program[5] := MOSTNEG INT    -- reserved for future
                                        -- use
        out.program[0] := MOSTNEG INT   -- standard i/o not used
        out.program[1] := MOSTNEG INT   -- standard i/o not used
        out.program[2] := MOSTNEG INT   -- reserved for future
                                        -- use
        LOAD.OUTPUT.CHANNEL (out.program[3], Pluto2Mars)
        LOAD.OUTPUT.CHANNEL (out.program[4], Pluto2Saturn)
        out.program[5] := MOSTNEG INT   -- reserved for future
                                        -- use
        -- Invoke the Ada program.
        -- Assumes the entry point name has been changed to
        -- "pluto.program".
        pluto.program (ws1, in.program, out.program, dummy.ws)
    :
```

2. PLUTOH2.OCC

```
#OPTION "AEV"

PROC pluto.program ([]INT ws1, in, out, ws2)
  [1000]INT d:
  SEQ
    SKIP
:
```

D. OCCAM HARNESSES ON PROCESSOR SATURN

1. SATURNH.OCC

```
#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC saturn.harness (CHAN OF INT Saturn2Venus, Venus2Saturn,
                     Saturn2Pluto, Pluto2Saturn,
                     []INT FreeMemory)

  #IMPORT "saturnh2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [6]INT in.program:
  [6]INT out.program:
```

```
      SEQ
        -- Set up vector of pointers to channels.
        in.program[0] := MOSTNEG INT      -- not used
        in.program[1] := MOSTNEG INT      -- standard i/o not used
        LOAD.INPUT.CHANNEL (in.program[2], Pluto2Saturn)
        LOAD.INPUT.CHANNEL (in.program[3], Venus2Saturn)
        in.program[4] := MOSTNEG INT      -- reserved for future
                                          -- use
        in.program[5] := MOSTNEG INT      -- reserved for future
                                          -- use
        out.program[0] := MOSTNEG INT     -- standard i/o not used
        out.program[1] := MOSTNEG INT     -- standard i/o not used
        LOAD.OUTPUT.CHANNEL (out.program[2], Saturn2Pluto)
        LOAD.OUTPUT.CHANNEL (out.program[3], Saturn2Venus)
        out.program[4] := MOSTNEG INT -- reserved for future use
        out.program[5] := MOSTNEG INT -- reserved for future use
        -- Invoke the Ada program.
        -- Assumes the entry point name has been changed to
        -- "saturn.program".
        saturn.program (ws1, in.program, out.program, dummy.ws)
    :
```

2. SATURNH2.OCC

```
#OPTION "AEV"

PROC saturn.program ([]INT ws1, in, out, ws2)
  [1000]INT d:
  SEQ
    SKIP
:
```

E. OCCAM HARNESSES ON PROCESSOR VENUS

1. VENUSH.OCC

```
#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC venus.harness (CHAN OF INT Mars2Venus, Venus2Mars,
                    Venus2Saturn, Saturn2Venus,
                    []INT FreeMemory)

  #IMPORT "venush2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [6]INT in.program:
```

```
  [6]INT out.program:
  SEQ
    -- Set up vector of pointers to channels.
    in.program[0] := MOSTNEG INT      -- not used
    in.program[1] := MOSTNEG INT      -- standard i/o not used
    LOAD.INPUT.CHANNEL (in.program[2], Mars2Venus)
    in.program[3] := MOSTNEG INT      -- not used
    in.program[4] := MOSTNEG INT      -- reserved for future
                                 -- use
    LOAD.INPUT.CHANNEL (in.program[5], Saturn2Venus)
    out.program[0] := MOSTNEG INT    -- standard i/o not used
    out.program[1] := MOSTNEG INT    -- standard i/o not used
    LOAD.OUTPUT.CHANNEL (out.program[2], Venus2Mars)
    out.program[3] := MOSTNEG INT -- reserved for future use
    out.program[4] := MOSTNEG INT -- resereved for future
                                 -- use
    LOAD.OUTPUT.CHANNEL (out.program[5], Venus2Saturn)
    -- Invoke the Ada program.
    -- Assumes the entry point name has been changed to
    -- "venus.program".
    venus.program (ws1, in.program, out.program, dummy.ws)
:
```

2.   VENUSH2.OCC

```
#OPTION "AEV"

PROC venus.program ([]INT ws1, in, out, ws2)
  [1000]INT d:
  SEQ
    SKIP                          .
:
```

APPENDIX B: COMMUNICATION LAYER/AUV FLOW ADA SOURCE CODE

A. COMMUNICATION LAYER ADA PROGRAMS

1. COMMON.ADA

```
with CHANNELS;
with CALENDAR;

package COMMON is

   -- Declarations of the statistics of the network and the
   -- common data types that are used in the communication
   -- scheme.

   NUM_PROGS  : constant INTEGER := 5 ;
   NUM_PATHS  : constant INTEGER := 13;
   NUM_ENTRYS : constant INTEGER := 19;

   type INT_16 is range -2**15 .. 2**15-1;

   type TASKS  is (HOST_TASK, TASK_SCREEN, TASK_AUTO_PILOT,
                   TASK_TIMER,TASK_VEHICLE_SYS,
                   TASK_EXE_MISSION, TASK_MONITOR,
                   TASK_AVOIDANCE, TASK_GUIDANCE,
                   TASK_NAVIGATION, TASK_SONAR,
                   EARTH_MAIN, MARS_MAIN, VENUS_MAIN,
                   SATURN_MAIN, PLUTO_MAIN,
                   NO_TASK, SHUTDOWN, LOOP_TASK);

   subtype EARTH_TASKS is TASKS range
        HOST_TASK..TASK_SCREEN;
   subtype MARS_TASKS is TASKS range
        TASK_AUTO_PILOT..TASK_VEHICLE_SYS;
   subtype PLUTO_TASKS is TASKS range
        TASK_EXE_MISSION..TASK_MONITOR;
   subtype SATURN_TASKS is TASKS range
        TASK_AVOIDANCE..TASK_GUIDANCE;
   subtype VENUS_TASKS is TASKS range
        TASK_NAVIGATION..TASK_SONAR;
   subtype MAIN_TASKS is TASKS range EARTH_MAIN..PLUTO_MAIN;
   subtype SPECIAL_TASKS is TASKS range NO_TASK..LOOP_TASK;


   type PROG_ARRAY is array (1..NUM_PROGS) of INTEGER;
```

```
type PATH_ARRAY is array (1..NUM_PATHS) of INTEGER;

type PROGRAMS     is (EARTH, MARS, VENUS, SATURN, PLUTO);

type ROUTING_STRATEGY is (CNT_CLK_RING, CLK_RING,
                    MULTI_PATH, BEST_PATH);


type OUT_TABLE is array(PROGRAMS range <>, NATURAL range
                  <>) of BOOLEAN;
type POINTER_TABLE is access OUT_TABLE;
type POINTER_CHANNEL is access CHANNELS.CHANNEL_ARRAY;


type COUNT_OUTPUTS is array(PROGRAMS range <>) of
                        NATURAL;
NR_OF_OUTPUTS: COUNT_OUTPUTS(MARS..PLUTO) := (3, others
                      => 2 );

type CONFIG_TABLE_MARS is array(ROUTING_STRATEGY) of
      OUT_TABLE(EARTH..PLUTO,0..NR_OF_OUTPUTS(MARS)-1);
MARS_CONFIG: CONFIG_TABLE_MARS :=
               ((-- counter clockwise ring
            -- OutToEarth  OutToVenus  OutToPluto
             (TRUE,        FALSE,       FALSE), -- EARTH
             (FALSE,       FALSE,       FALSE), -- MARS
             (FALSE,       FALSE,       TRUE),  -- VENUS
             (FALSE,       FALSE,       TRUE),  -- SATURN
             (FALSE,       FALSE,       TRUE)), -- PLUTO


               (-- clockwise ring
            -- OutToEarth  OutToVenus  OutToPluto
             (TRUE,        FALSE,       FALSE),  -- EARTH
             (FALSE,       FALSE,        FALSE),   -- MARS
             (FALSE,       TRUE,        FALSE),  -- VENUS
             (FALSE,       TRUE,        FALSE),  -- SATURN
             (FALSE,       TRUE,        FALSE)), -- PLUTO


               (-- Multipath
            -- OutToEarth  OutToVenus  OutToPluto
             (TRUE,        FALSE,       FALSE), --  EARTH
             (FALSE,       FALSE,       FALSE), --  MARS
             (FALSE,       TRUE,        FALSE), --  VENUS
             (FALSE,       TRUE,        TRUE),  --  SATURN
             (FALSE,       FALSE,       TRUE)), --  PLUTO


               (-- Bestpath
            --  OutToEarth  OutToVenus  OutToPluto
             (TRUE,        FALSE,       FALSE), -- EARTH
             (FALSE,       FALSE,       FALSE), -- MARS
             (FALSE,       TRUE,        FALSE), -- VENUS
```

```
                   (FALSE,        TRUE,         FALSE), -- SATURN
                   (FALSE,        FALSE,        TRUE))); -- PLUTO

    type CONFIG_TABLE_PLUTO is array(ROUTING_STRATEGY) of
          OUT_TABLE(EARTH..PLUTO,0..NR_OF_OUTPUTS(PLUTO)-1);
    PLUTO_CONFIG: CONFIG_TABLE_PLUTO :=
                   ((-- counter clockwise ring
                    --OutToMars  OutToSaturn
                     (FALSE,        TRUE),     -- EARTH
                     (FALSE,        TRUE),     -- MARS
                     (FALSE,        TRUE),     -- VENUS
                     (FALSE,        TRUE),     -- SATURN
                     (FALSE,        FALSE)),   -- PLUTO

                    (-- clockwise ring
                     --OutToMars  OutToSaturn
                     (TRUE,         FALSE),    -- EARTH
                     (TRUE,         FALSE),    -- MARS
                     (TRUE,         FALSE),    -- VENUS
                     (TRUE,         FALSE),    -- SATURN
                     (FALSE,        FALSE)),   -- PLUTO

                    (-- Multipath
                     --OutToMars  OutToSaturn
                     (TRUE,         FALSE),    -- EARTH
                     (TRUE,         FALSE),    -- MARS
                     (TRUE,         TRUE),     -- VENUS
                     (FALSE,        TRUE),     -- SATURN
                     (FALSE,        FALSE)),   -- PLUTO

                    (-- Bestpath
                     --OutToMars  OutToSaturn
                     (TRUE,         FALSE),    -- EARTH
                     (TRUE,         FALSE),    -- MARS
                     (FALSE,        TRUE),     -- VENUS
                     (FALSE,        TRUE),     -- SATURN
                     (FALSE,        FALSE)));  -- PLUTO

    type CONFIG_TABLE_SATURN is array(ROUTING_STRATEGY) of
          OUT_TABLE(EARTH..PLUTO,0..NR_OF_OUTPUTS(SATURN)-1);
    SATURN_CONFIG: CONFIG_TABLE_SATURN :=

                   (( -- counter clockwise ring
                   -- OutToVenus OutToPluto
                     (TRUE,         FALSE),    -- EARTH
                     (TRUE,         FALSE),    -- MARS
                     (TRUE,         FALSE),    -- VENUS
                     (FALSE,        FALSE),    -- SATURN
                     (TRUE,         FALSE )),  -- PLUTO
```

84

```
                    ( -- clockwise ring
                -- OutToVenus  OutToPluto
                   (FALSE,       TRUE ),  -- EARTH
                   (FALSE,       TRUE),   -- MARS
                   (FALSE,       TRUE),   -- VENUS
                   (FALSE,       FALSE),  -- SATURN
                   (FALSE,       TRUE)),  -- PLUTO


                    ( -- Multipath
                -- OutToVenus  OutToPluto
                   (TRUE,        TRUE),   -- EARTH
                   (TRUE,        TRUE),   -- MARS
                   (TRUE,        FALSE),  -- VENUS
                   (FALSE,       FALSE),  -- SATURN
                   (FALSE,        TRUE)), -- PLUTO

                    ( -- Bestpath
                -- OutToVenus OutToPluto
                   (FALSE,       TRUE),   -- EARTH
                   (FALSE,       TRUE),   -- MARS
                   (TRUE,        FALSE),  -- VENUS
                   (FALSE,       FALSE),  -- SATURN
                   (FALSE,       TRUE))); -- PLUTO

  type CONFIG_TABLE_VENUS is array(ROUTING_STRATEGY) of
         OUT_TABLE(EARTH..PLUTO,0..NR_OF_OUTPUTS(VENUS)-1);
  VENUS_CONFIG: CONFIG_TABLE_VENUS :=

                   ((-- counter clockwise ring
                   -- OutToMars, OutToSaturn
                   (TRUE,        FALSE), -- EARTH
                   (TRUE,        FALSE), -- MARS
                   (FALSE,       FALSE), -- VENUS
                   (TRUE,        FALSE),  -- SATURN
                   (TRUE,        FALSE)), -- PLUTO

                   (-- clockwise ring
                   -- OutToMars, OutToSaturn
                   (FALSE,       TRUE),  -- EARTH
                   (FALSE,       TRUE),  -- MARS
                   (FALSE,       FALSE), -- VENUS
                   (FALSE,       TRUE),   -- SATURN
                   (FALSE,       TRUE)),  -- PLUTO

                   (-- Multipath
                   -- OutToMars, OutToSaturn
                   (TRUE,        FALSE), -- EARTH
                   (TRUE,        FALSE), -- MARS
                   (FALSE,       FALSE), -- VENUS
                   (FALSE,       TRUE),   -- SATURN
```

```
                (TRUE,          TRUE)),  -- PLUTO

                (-- Bestpath
                -- OutToMars, OutToSaturn
                (TRUE,          FALSE),  -- EARTH
                (TRUE,          FALSE),  -- MARS
                (FALSE,         FALSE),  -- VENUS
                (FALSE,         TRUE),   -- SATURN
                (TRUE,          FALSE))); -- PLUTO


    type ENTRYS is (OUTPUT, UPDATE_SONAR, VS_ORDERS,
                SYS_STATUS,AP_ORDERS,UPDATE_NAV,
                UPDATE_ORDERS, AVOID_REC, SONAR_OBSTICLE,
                OBJECT_ALERT, EXE_UPDATE, OB_AVOID,
                MONITOR_UPDATE, TO_MONITOR, PILOT_UPDATE,
                ACKNOWLEGE, NO_ENT, RETURNING,
                TEST_TIME);

    type MESSAGE_FORM is
        record
            ORIGIN        : TASKS  := NO_TASK;
            DESTIN        : TASKS  := NO_TASK;
            ENT_CALL      : ENTRYS := NO_ENT;
            TIME_STAMP    : DURATION := 0.0;
            CODE_1        : INT_16 := 0;
            CODE_2        : INT_16 := 0;
            MESSAGE_CODE  : INT_16 := 0;
            PROG          : PROG_ARRAY := (others =>0);
            PATH          : PATH_ARRAY := (others =>0);
        end record;

    type MASK_TYPE is array(MARS..PLUTO) of BOOLEAN;

    type CONFIG_MESSAGE is
        record
          MASK          : MASK_TYPE := (others => FALSE);
          ROUT_INFO     : ROUTING_STRATEGY;
        end record;

    type PERIOD_MESSAGE is
        record
          MASK          : MASK_TYPE := (others => FALSE);
          PERIOD_INFO : DURATION;
        end record;
```

86

```
    SHUTDOWN_MESSAGE : MESSAGE_FORM := (SHUTDOWN, SHUTDOWN,
        NO_ENT, 0.0, 0, 0, 0, (others => 0), (others => 9));

    HOST              : constant PROGRAMS := EARTH;

    READ_INT          : constant DURATION := 5.0;

    -- Instantiations of the generic channel i/o package.

    package MESSAGE_IO is new CHANNELS.CHANNEL_IO
    (MESSAGE_FORM);
    package CONFIG_IO  is new CHANNELS.CHANNEL_IO
    (CONFIG_MESSAGE);
    package PERIOD_IO  is new CHANNELS.CHANNEL_IO
    (PERIOD_MESSAGE);

    function IF_ITS_HERE (FROM_PROGRAM : in PROGRAMS; TO_TASK
    : in TASKS) return BOOLEAN;

    function WHERE_IS    (IN_TASK: in TASKS) return PROGRAMS;

end COMMON;

package body COMMON is

    function IF_ITS_HERE (FROM_PROGRAM : in PROGRAMS; TO_TASK
    : in TASKS) return BOOLEAN is


    begin
      case FROM_PROGRAM is
        when EARTH =>
          case TO_TASK is
            when EARTH_TASKS'FIRST..EARTH_TASKS'LAST =>
              return TRUE;
            when others =>
              return FALSE;
          end case;
        when MARS =>
          case TO_TASK is
            when MARS_TASKS'FIRST..MARS_TASKS'LAST =>
              return TRUE;
            when others =>
              return FALSE;
          end case;
        when PLUTO =>
          case TO_TASK is
            when PLUTO_TASKS'FIRST..PLUTO_TASKS'LAST |
              LOOP_TASK   =>
              return TRUE;
            when others =>
```

```ada
            return FALSE;
          end case;
      when SATURN =>
        case TO_TASK is
          when SATURN_TASKS'FIRST..SATURN_TASKS'LAST =>
            return TRUE;
          when others =>
            return FALSE;
        end case;
      when VENUS =>
         case TO_TASK is
           when VENUS_TASKS'FIRST..VENUS_TASKS'LAST =>
             return TRUE;
           when others =>
             return FALSE;
         end case;
      when others =>
         return FALSE;
    end case;

  end IF_ITS_HERE;

  function WHERE_IS(IN_TASK: in TASKS) return PROGRAMS is

  begin
    case IN_TASK  is
      when EARTH_TASKS'FIRST..EARTH_TASKS'LAST =>
        return EARTH;
      when MARS_TASKS'FIRST..MARS_TASKS'LAST =>
        return MARS;
      when PLUTO_TASKS'FIRST..PLUTO_TASKS'LAST | LOOP_TASK
               =>
        return PLUTO;
      when SATURN_TASKS'FIRST..SATURN_TASKS'LAST =>
        return SATURN;
      when VENUS_TASKS'FIRST..VENUS_TASKS'LAST =>
        return VENUS;
      when others =>
        return MARS;
    end case;
  end WHERE_IS;
end COMMON;
```

2.   COMLAYER.ADA

```ada
with COMMON;
use  COMMON;
with CALENDAR;
```

```ada
use  CALENDAR;
with CHANNELS;
with RANDOM;


generic

  --Function to be instantiated on each node program.
  --This function issues entry calls that send messages to
  --local tasks.
  with procedure SEND_IT (MESSAGE: in MESSAGE_FORM;
                               MESS: out BOOLEAN);

package COMLAYER is

  --Local mailman implemented by a circular buffer.Puts the
  --local arriving messages in a circular buffer and
  --distribute them to the local tasks.
  task QUE is
    -- Called by the task INOUT to pass a local message.
    entry TO_QUE (QUE_MESSAGE: in MESSAGE_FORM);
  end;

  --Traffic handler: Sends the local messages to task QUE
  --and sends the external messages to the remote nodes
  --through appropriate output channels.
  task INOUT is
    entry INIT1     (SITE            : in PROGRAMS);
    entry INIT2     (SEND_ARRAY  :
                        in CHANNELS.CHANNEL_ARRAY);
    entry INIT3     (SEND_TABLE : in OUT_TABLE);
    entry INCOMING (INOUT_MESSAGE : in MESSAGE_FORM);
    entry SEND      (INOUT_MESSAGE : in MESSAGE_FORM);
  end;
end COMLAYER;


package body COMLAYER is

  --Local mailman implemented by a circular buffer.Puts the
  --local arriving messages in a circular buffer and
  --distribute them to the local tasks.
  task body QUE is
    MAX_STORAGE      : constant INTEGER := 20;
    --Circular buffer size
    SENT_MESSAGE    : BOOLEAN              := FALSE;
    --Indicates if the rendevouz for local message delivery
    --was accomplished succssefully.
    ALL_FULL         : BOOLEAN              := FALSE;
    --Indicates that the circular buffer is full. That
    --is,there is no more room for an incoming message.
```

```
      PEND_MESS        : BOOLEAN              := FALSE;
      --Indicates that there is  a pending message waiting
      --for a position in the circular buffer.
      FULL             : constant BOOLEAN := TRUE;
      --Indicates that the current position of the circular
      --buffer contains valid message which is still
      --to be delivered.
      EMPTY            : constant BOOLEAN := FALSE;
      --Indicates that the current position of the circular
      --buffer has a non valid message. That is, this
      --position is empty or this message has already been
      --delivered and can be discarded.
      NUMBER           : INTEGER              := 0; -- Index that
      --points to a circular buffer position.
      MESSAGES_IN_MAIL: INTEGER              := 0; -- Control the
      --number of messages in the circular buffer.
      SLOT:    array(0 .. (MAX_STORAGE-1)) of BOOLEAN :=
      (others => FALSE);
      --SLOT indicates if the pointed circular buffer
      --position contains a valid message or not.
      STORAGE          : array(0 .. (MAX_STORAGE-1)) of
      MESSAGE_FORM;
      --STORAGE is the array that implements the circular
      --buffer.
      TEMP_MESSAGE     : MESSAGE_FORM; -- temporary variable to
      --hold the incoming message.

   begin
     MAIN: loop
       select
         --Accept calls from the task INOUT with arriving
         --local messages
         accept TO_QUE (QUE_MESSAGE : in MESSAGE_FORM) do
           TEMP_MESSAGE := QUE_MESSAGE;
         end TO_QUE;

         -- Puts the message into the circular buffer
         STORAGE (NUMBER) := TEMP_MESSAGE;
         MESSAGES_IN_MAIL := MESSAGES_IN_MAIL + 1;
         SLOT (NUMBER) := FULL;

         --Priority is given to any messages waiting to be
         --mailed, so another accept statement is needed
         --before attempting to deliver a message.
         SEND:loop
           if ALL_FULL = FALSE then -- the circular buffer
             --has room for a new incoming message.
             select
               --Accept calls from the task INOUT with
               --arriving local messages.
```

90

```
accept TO_QUE (QUE_MESSAGE : in MESSAGE_FORM)
do TEMP_MESSAGE := QUE_MESSAGE;
end TO_QUE;

if MESSAGES_IN_MAIL < MAX_STORAGE then   -- If
   --the circular buffer has room for a new
   --incoming message searches for the next
   --free position in the circular buffer
   --and stores the message in this position.
   STORE: loop
      if SLOT(NUMBER) = EMPTY then
         --Free position was found.
         --Puts the message into the circular
         --buffer.
         STORAGE(NUMBER) := TEMP_MESSAGE;
         MESSAGES_IN_MAIL := MESSAGES_IN_MAIL+1;
         SLOT(NUMBER) := FULL;
         exit;
      end if;
      --Add 1 to NUMBER so that next mail slot
      --can be checked.
      NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
   end loop STORE;


   --Searches for the next used position in the
   --circular buffer.
   if MESSAGES_IN_MAIL /= 0 then
      NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
      while SLOT(NUMBER) /= FULL loop
         NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
      end loop;
   end if;

else
   --the circular buffer has no room for a
   --new incoming message.
   ALL_FULL := TRUE;
   PEND_MESS := TRUE;
end if;
or
   --cannot accept a new message from task INOUT
   --so suspends the task and puts the task at
   --the end of the ready queue.
   delay 0.0;
end select;
end if;

--if the circular buffer position contains a valid
--message call SEND_IT in order to issue an entry
--call for message delivery.
```

```
          if SLOT(NUMBER) = FULL then
            SEND_IT (STORAGE (NUMBER), SENT_MESSAGE);
          end if;

          --if a rendevouz with the destination task has
          --ocurred then the message was successfully sent
          --to its destination.
          if SENT_MESSAGE then
            SENT_MESSAGE := FALSE; -- turn off the flag
            SLOT(NUMBER) := EMPTY; -- liberates circular
            -- buffer position
            MESSAGES_IN_MAIL := MESSAGES_IN_MAIL - 1;

            if PEND_MESS then --if there is a pending
            --message store the pending message in the
            --just freed circular buffer position
              STORAGE(NUMBER) := TEMP_MESSAGE;
              SLOT(NUMBER) := FULL;
              MESSAGES_IN_MAIL := MESSAGES_IN_MAIL + 1;
              PEND_MESS := FALSE;
              NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
            end if;

            --updates the status of the boolean variable
            --ALL_FULL which indicates whether the circular
            --buffer is full or not
            if MESSAGES_IN_MAIL < MAX_STORAGE then
              ALL_FULL := FALSE;
            else
              ALL_FULL := TRUE;
            end if;

          else --The message was not succssefully sent so it
            --loses its turn and has to wait for the next
            --circular buffer scan
            NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
          end if;

          --To save processor time the loop is exited when
          --there are no pending mail deliveries.
          exit when MESSAGES_IN_MAIL = 0;
        end loop SEND;
      or
        terminate;
      end select;
    end loop MAIN;
end QUE;
```

```
--Traffic handler: Sends the local messages to task QUE
--and sends the external messages to the remote nodes
--through appropriate output channels.
task body INOUT is

  HERE              : BOOLEAN; -- indicates if the message is
  --destinated to the local node or to an external node.
  STORAGE_MESSAGE: MESSAGE_FORM; --message just arrived
  --in the traffic handler
  CURRENT_NODE       : PROGRAMS; --local processor ID
  OUT_CHANNEL        : CHANNELS.CHANNEL_REF; --it will
  --point to the output channel to be writen by the task
  --INOUT when sending a message to a external processor.
  DEST_PROGRAM       : PROGRAMS; --holds the processor ID
  --of the message destination
  --The IND_INDEX array is used to hold all the possible
  --indexes that points to the output channels which can
  --be used to send messages to a specific processor.
  --These indexes can reference the output channels when
  --used with CURRENT_ARRAY
  type INDEX_ARRAY is ARRAY(NATURAL range <>) of NATURAL;
  IND_INDEX: INDEX_ARRAY(0..3); --Indirect index array for
  --the output channels
  OUT_COUNT: NATURAL; --counter used as index for the
  --IND_INDEX array
  TOSS:   FLOAT; -- random variable that is used when we
  --have more than one possible output channel to send a
  --message to a specific  processor.
  CURRENT_ARRAY : POINTER_CHANNEL; -- It is an acess type
  --variable which type is defined in the package COMMON.
  --CURRENT_ARRAY points to the array of output channels
  --to be used by the task INOUT.
  CURRENT_TABLE : POINTER_TABLE;  --It is an acess type
  --variable which type is defined in the package COMMON.
  --CURRENT_TABLE points to the rounting table to be used
  --by the rounting algorithm.
  ARRIVED         : BOOLEAN := FALSE;

begin
  -- Accept intialization messages from the main processor
  --program
  accept INIT1 (SITE  : in PROGRAMS) do
    CURRENT_NODE := SITE;         -- local processor ID
  end INIT1;

  accept INIT2 (SEND_ARRAY : in CHANNELS.CHANNEL_ARRAY) do
    CURRENT_ARRAY := new
    CHANNELS.CHANNEL_ARRAY'(SEND_ARRAY); -- output
    --channels array
  end INIT2;
```

93

```
accept INIT3 (SEND_TABLE: in OUT_TABLE) do
  CURRENT_TABLE := new OUT_TABLE'(SEND_TABLE);
  --routing table
end INIT3;

loop
  select
    --Used to accept messages just arrived from
    --external nodes.Called by each main node program.
    accept INCOMING (INOUT_MESSAGE : in MESSAGE_FORM) do
      STORAGE_MESSAGE := INOUT_MESSAGE;
      ARRIVED := TRUE;
    end INCOMING;
  or
    --Used to accept messages from local tasks.
    accept SEND (INOUT_MESSAGE: in MESSAGE_FORM) do
      STORAGE_MESSAGE := INOUT_MESSAGE;
      ARRIVED := TRUE;
    end SEND;
  or
    --terminate;
    delay 0.001;
  end select;

  if ARRIVED then
    ARRIVED := FALSE;
    OUT_COUNT := 0; --initialize counter used as index
    --for the IND_INDEX array
    --Find the destination processor by using the
    --function WHERE_IS which
    --is declared in the package COMMON
    DEST_PROGRAM := WHERE_IS (STORAGE_MESSAGE.DESTIN);
    --Verifies if the message is for the local node or
    --for an external processor
    if DEST_PROGRAM = CURRENT_NODE then
      HERE := TRUE;
    else
      HERE := FALSE;
    end if;
    if HERE then -- if destination is the local node it
      --passes the message to the local mailman
    QUE.TO_QUE (STORAGE_MESSAGE);
    else
      --Destination is an external processor
      --Search for all output channels that can be used
      --to communicate with the destination processor
      --and store this information in the IND_INDEX
      --array
      for J in CURRENT_TABLE.all'RANGE(2) loop
        if CURRENT_TABLE(DEST_PROGRAM, J) = TRUE then
          IND_INDEX(OUT_COUNT) := J;
```

94

e. *RAN_INT.ADA*

```
separate (RANDOM)

function RANDOM_INT (N : POSITIVE) return POSITIVE is
  RESULT : INTEGER range 1 .. N;
begin
  RESULT := INTEGER (FLOAT (N) * UNIT_RANDOM + 0.5);
  return RESULT;
exception
  when CONSTRAINT_ERROR ¦ NUMERIC_ERROR =>
    -- If machine rounds 0.5 down to 0, return 1.
    return 1;
end RANDOM_INT;
```

B. HOST ADA PROGRAMS

1. HOSTLAY.ADA

```
with COMMON;
use  COMMON;


generic

  --Function to be instantiated
  with procedure SEND_IT (MESSAGE: in MESSAGE_FORM;
                               ACK: out BOOLEAN;
                               MESS: out BOOLEAN);

package HOST_LAYER is

  task EARTH_QUE is
    entry TO_QUE (QUE_MESSAGE: in MESSAGE_FORM);
  end;

end HOST_LAYER;



package body HOST_LAYER is

  task body EARTH_QUE is

    MAX_STORAGE      : constant INTEGER := 20;
    SENT_MESSAGE     : BOOLEAN              := FALSE;
    SENT_ACK         : BOOLEAN              := FALSE;
    ALL_FULL         : BOOLEAN              := FALSE;
```

97

```
        PEND_MESS           : BOOLEAN              := FALSE;
        FULL                : constant BOOLEAN := TRUE;
        EMPTY               : constant BOOLEAN := FALSE;
        NUMBER              : INTEGER              := 0;
        MESSAGES_IN_MAIL: INTEGER                := 0;
        SLOT: array(0 .. (MAX_STORAGE-1)) of BOOLEAN := (others
        => FALSE);
        STORAGE: array(0 .. (MAX_STORAGE-1)) of MESSAGE_FORM;
        TEMP_MESSAGE      : MESSAGE_FORM;

    begin
      MAIN: loop
        select
          accept TO_QUE (QUE_MESSAGE : in MESSAGE_FORM) do
            TEMP_MESSAGE := QUE_MESSAGE;
          end TO_QUE;

          STORAGE (NUMBER) := TEMP_MESSAGE;
          MESSAGES_IN_MAIL := MESSAGES_IN_MAIL + 1;
          SLOT (NUMBER) := FULL;

          --Priority is given to any messages waiting to be
          --mailed, so another ACCEPT statement is needed
          --before attempting to deliver a message.

          SEND:loop
            if ALL_FULL = FALSE then
              select
                accept TO_QUE (QUE_MESSAGE : in MESSAGE_FORM)
                                            do
                  TEMP_MESSAGE := QUE_MESSAGE;
                end TO_QUE;

                if MESSAGES_IN_MAIL < MAX_STORAGE then
                  STORE: loop
                    if SLOT(NUMBER) = EMPTY then
                      STORAGE(NUMBER) := TEMP_MESSAGE;
                      MESSAGES_IN_MAIL := MESSAGES_IN_MAIL+1;
                      SLOT(NUMBER) := FULL;
                      exit;
                    end if;
                    --Add 1 to NUMBER so that next mail slot
                    --can be checked.
                    NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
                  end loop STORE;

                  --Add 1 to NUMBER so that last in will not
                  --be forst out if there are other messages
                  --in the queue
                  if MESSAGES_IN_MAIL /= 0 then
                    NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
```

```
                  while SLOT(NUMBER) /= FULL loop
                    NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
                  end loop;
               end if;

               --This is a flag that says that are
               --incoming messages not yet stored in
               --the queue, and no others should be read
               --until it is.

             else
               ALL_FULL := TRUE;
               PEND_MESS := TRUE;
             end if;
           or
             delay 0.0;
           end select;
        end if;
        if SLOT(NUMBER) = FULL then
          SEND_IT (STORAGE (NUMBER), SENT_ACK,
          SENT_MESSAGE);
        end if;

        if SENT_MESSAGE then
          SENT_MESSAGE := FALSE;
          SLOT(NUMBER) := EMPTY;
          MESSAGES_IN_MAIL := MESSAGES_IN_MAIL - 1;

          if  PEND_MESS then
            STORAGE(NUMBER) := TEMP_MESSAGE;
            SLOT(NUMBER) := FULL;
            MESSAGES_IN_MAIL := MESSAGES_IN_MAIL + 1;
            PEND_MESS    := FALSE;
            NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
          end if;

          if MESSAGES_IN_MAIL < MAX_STORAGE then
            ALL_FULL := FALSE;
          else
            ALL_FULL := TRUE;
          end if;
        else
          NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
        end if;
        -- To save processor time the loop is exited when
        -- there are no pending
        -- mail deliveries.

        exit when MESSAGES_IN_MAIL = 0;
      end loop SEND;
   or
```

```
          terminate;
        end select;
      end loop MAIN;

   end EARTH_QUE;
end HOST_LAYER;
```

   2.   EARTH.ADA

```
with TEXT_IO;
with COMMON;
use COMMON;
with PRINTOUT;
use PRINTOUT;
with HOST_LAYER;
with CHANNELS;
with CALENDAR;
use  CALENDAR;

procedure EARTH is

   package INTEGER_INOUT is new TEXT_IO.INTEGER_IO(INTEGER);
   use INTEGER_INOUT;
   package TIME_IO is new TEXT_IO.FIXED_IO(DURATION);
   package FLT_IO is new TEXT_IO.FLOAT_IO(FLOAT);

   CONFIG_OPTION : INTEGER;
   IN_MESSAGE   : MESSAGE_FORM;
   MAIN_TALK    : MESSAGE_FORM;
   FIRST_MESSAGE: CONFIG_MESSAGE;
   SECOND_MESSAGE: PERIOD_MESSAGE;
   LOCATION     : constant PROGRAMS := EARTH;
   TIME_OUT     : TIME;
   QUIT_TIME    : TIME;
   ABORTED      : BOOLEAN;
   FAILED       : INT_16 := 0;
   MESS_COUNT   : INT_16 := 0;
   QUIT_INT     : DURATION := 55.0;
   PERIOD       : DURATION;
   MESSAGE_ERROR: exception;

   InFromMars: CHANNELS.CHANNEL_REF :=
   CHANNELS.IN_PARAMETERS (2);
   OutToMars:  CHANNELS.CHANNEL_REF :=
   CHANNELS.OUT_PARAMETERS(2);

   task SCREEN is
     entry OUTPUT (SCREEN_MESSAGE: in MESSAGE_FORM);
   end;
```

```
procedure SEND_IT_FROM_EARTH (MESSAGE: in MESSAGE_FORM;
                              ACK     : out BOOLEAN;
                              MESS    : out BOOLEAN) is

   MESSAGE_SENT: BOOLEAN := FALSE;
   ACK_SENT    : BOOLEAN := FALSE;

begin
   select
      SCREEN.OUTPUT (MESSAGE);
      MESSAGE_SENT := TRUE;
   or
      delay 0.01;
   end select;
   ACK := ACK_SENT;
   MESS := MESSAGE_SENT;
   return;
end SEND_IT_FROM_EARTH;

package EARTH_LAYER is new HOST_LAYER(SEND_IT_FROM_EARTH);
use EARTH_LAYER;


task body SCREEN is

   OUT2SCREEN: MESSAGE_FORM;
   LOCALS    : array (TASKS) of INT_16 := (others => 0);
   COUNT     : INTEGER := 0;
   N         : INTEGER := 0;
   AVE_TIME  : FLOAT;
   START_STAMP: CALENDAR.TIME;
   TIMER     : DURATION := 0.0;
   TOT_TIME  : DURATION := 0.0;
   OUT_TIME  : DURATION := 0.0;

begin

   MAIN: loop
      accept OUTPUT (SCREEN_MESSAGE: in MESSAGE_FORM) do
         OUT2SCREEN := SCREEN_MESSAGE;
      end OUTPUT;

      case OUT2SCREEN.MESSAGE_CODE is
         when 11 =>
            TEXT_IO.PUT_LINE ("Main Earth program finished.");
            PRINT_MESSAGE(OUT2SCREEN);
            exit;
         when 20 =>
            LOCALS (OUT2SCREEN.ORIGIN) :=
            LOCALS(OUT2SCREEN.ORIGIN) + 1;
```

```
              TOT_TIME := TOT_TIME + OUT2SCREEN.TIME_STAMP;
              N := N + 1;
          when 21 =>
             LOCALS(OUT2SCREEN.ORIGIN) :=
             LOCALS(OUT2SCREEN.ORIGIN) + 1;
          when 30 =>
             START_STAMP := CLOCK;
          when 31 =>
             TIMER := CLOCK - START_STAMP;
             OUT_TIME := OUT2SCREEN.TIME_STAMP;
          when 71 =>
             TEXT_IO.PUT_LINE(" Vehicle Sys ");
          when 72 =>
             TEXT_IO.PUT_LINE(" Sonar ");
          when 73 =>
             TEXT_IO.PUT_LINE(" Navigation ");
          when 74 =>
             TEXT_IO.PUT_LINE(" Monitor ");
          when 75 =>
             TEXT_IO.PUT_LINE(" Avoidance ");
          when 76 =>
             TEXT_IO.PUT_LINE(" Exe Mission ");
          when 77 =>
             TEXT_IO.PUT_LINE(" Guidance ");
          when 78 =>
             TEXT_IO.PUT_LINE(" Auto Pilot ");
          when 99 =>
             TEXT_IO.PUT_LINE("Shutdown Received.");
          when others =>
             TEXT_IO.PUT_LINE ("Bad MESSAGE_CODE.");
       end case;
     end loop MAIN;

  TEXT_IO.PUT ("EARTH_MAIN = ");
  INT_IO.PUT (LOCALS(EARTH_MAIN));
  TEXT_IO.NEW_LINE;

  TEXT_IO.PUT("TASK_VEHICLE_SYS = ");
  INT_IO.PUT(LOCALS(TASK_VEHICLE_SYS));
  TEXT_IO.NEW_LINE;

  TEXT_IO.PUT ("Total time from SCREEN was : ");
  TIME_IO.PUT (OUT_TIME);
  TEXT_IO.NEW_LINE;

  TEXT_IO.PUT ("Ave Time calculated from VEHICLE_SYS was :
  ");
  AVE_TIME := FLOAT(TOT_TIME) / FLOAT(N);
  FLT_IO.PUT(AVE_TIME);
  TEXT_IO.NEW_LINE;
end SCREEN;
```

```ada
procedure PRINT_HEADER is

begin
  TEXT_IO.PUT_LINE("Please, Choose one of options(1 to 4)
  below");
  TEXT_IO.PUT_LINE("          1)Ring Counter Clockwise");
  TEXT_IO.PUT_LINE("          2)Ring Clockwise");
  TEXT_IO.PUT_LINE("          3)Random Multipath");
  TEXT_IO.PUT_LINE("          4)Best Path");
end PRINT_HEADER;

procedure PRINT_QUERY is

begin
  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT_LINE("Please, Enter the repetition period
  interval.");
end PRINT_QUERY;

function CHECK_MASK(MASK: MASK_TYPE) return BOOLEAN is

begin
  for I in MASK'range loop
    if MASK(I) = FALSE then
      return FALSE;
    end if;
  end loop;
  return TRUE;
end CHECK_MASK;

-- main program
begin
  PRINT_HEADER;
  GET(CONFIG_OPTION);
  case CONFIG_OPTION is
    when 1 =>
      FIRST_MESSAGE.ROUT_INFO := CNT_CLK_RING;
    when 2 =>
      FIRST_MESSAGE.ROUT_INFO := CLK_RING;
    when 3 =>
      FIRST_MESSAGE.ROUT_INFO := MULTI_PATH;
    when 4 =>
      FIRST_MESSAGE.ROUT_INFO := BEST_PATH;
    when others =>
      raise CONSTRAINT_ERROR;
  end case;
  PRINT_QUERY;
  TIME_IO.GET(SECOND_MESSAGE.PERIOD_INFO);

  CONFIG_IO.WRITE(OutToMars, FIRST_MESSAGE);
  CONFIG_IO.READ(InFromMars, FIRST_MESSAGE);
```

```
      if CHECK_MASK(FIRST_MESSAGE.MASK) = FALSE then
        raise MESSAGE_ERROR;
      end if;

      PERIOD_IO.WRITE(OutToMars, SECOND_MESSAGE);
      PERIOD_IO.READ(InFromMars,  SECOND_MESSAGE);
      if CHECK_MASK(SECOND_MESSAGE.MASK) = FALSE then
        raise MESSAGE_ERROR;
      end if;



      QUIT_TIME := CLOCK + QUIT_INT;
      MAIN_TALK.DESTIN  := TASK_SCREEN;
      MAIN_TALK.ORIGIN  := EARTH_MAIN;
      MAIN_TALK.MESSAGE_CODE := 21;
      EARTH_QUE.TO_QUE (MAIN_TALK);

      loop
        TIME_OUT := CLOCK + READ_INT;
        MESSAGE_IO.READ_OR_FAIL (InFromMars, IN_MESSAGE,
      TIME_OUT, ABORTED);
        if ABORTED then
          FAILED := FAILED + 1;
        else
          MESS_COUNT := MESS_COUNT + 1;
          IN_MESSAGE.PROG(1) := IN_MESSAGE.PROG(1) + 1;
          EARTH_QUE.TO_QUE(IN_MESSAGE);
        end if;

        exit when CLOCK > QUIT_TIME;
      end loop;

      MAIN_TALK := IN_MESSAGE;
      MAIN_TALK.DESTIN := TASK_SCREEN;
      MAIN_TALK.MESSAGE_CODE := 11;
      MAIN_TALK.CODE_1 := FAILED;
      MAIN_TALK.CODE_2 := MESS_COUNT;
      EARTH_QUE.TO_QUE (MAIN_TALK);

end EARTH;
```

3. PRINTOUT.ADA

```
with COMMON;
use  COMMON;
with TEXT_IO;
package PRINTOUT is
  package PRINT_TASK  is new
  TEXT_IO.ENUMERATION_IO (TASKS)    ;
```

```ada
   package PRINT_PROG  is new TEXT_IO.ENUMERATION_IO
   (PROGRAMS);
   package INT_IO is new TEXT_IO.INTEGER_IO(INT_16);
   procedure PRINT_MESSAGE (MESSAGE : in MESSAGE_FORM);
end PRINTOUT;

package body PRINTOUT is

   procedure PRINT_MESSAGE (MESSAGE : in MESSAGE_FORM) is
      TO_TASK_NAME    : TASKS  ;
      FROM_TASK_NAME : TASKS  ;
      I              : INTEGER;

   begin
      FROM_TASK_NAME := MESSAGE.ORIGIN     ;
      TEXT_IO.NEW_LINE;
      TEXT_IO.PUT_LINE
      ("************************************************
      **********");
      TEXT_IO.PUT_LINE ("*                       Message Report
                *");
      TEXT_IO.PUT_LINE ("*
                *");
      TEXT_IO.PUT        ("*  From        : ");
      PRINT_TASK.PUT (FROM_TASK_NAME,37) ;
      TEXT_IO.PUT_LINE (" *");
      TEXT_IO.PUT_LINE ("*                       Path Array:
                *");
      TEXT_IO.PUT ("*");

      for I in 1..NUM_PATHS loop
        TEXT_IO.PUT (" ");
        INT_IO.PUT (INT_16 (MESSAGE.PATH(I)),3);
      end loop;
      TEXT_IO.PUT_LINE (" *");
      TEXT_IO.PUT_LINE ("*                       Program Array:
                *");
      TEXT_IO.PUT ("*                    ");

      for I in 1..NUM_PROGS loop
        TEXT_IO.PUT (" ");
        INT_IO.PUT  (INT_16 (MESSAGE.PROG(I)),3);
        TEXT_IO.PUT (" ");
      end loop;

      TEXT_IO.PUT_LINE ("                    *");
      TEXT_IO.PUT        ("* CODE_1 : ");
      INT_IO.PUT        (MESSAGE.CODE_1,3);
      TEXT_IO.PUT        ("   CODE_2 : ");
      INT_IO.PUT (MESSAGE.CODE_2,3);
      TEXT_IO.PUT        ("  Message Code : ");
```

```
            INT_IO.PUT (MESSAGE.MESSAGE_CODE);
            TEXT_IO.PUT_LINE ("   *");
            TEXT_IO.PUT_LINE
            ("*********************************
            ********************");
      end PRINT_MESSAGE;

end PRINTOUT;
```

C. AUV FLOW MAIN ADA PROGRAMS

1. MARS.ADA

```
with COMMON;
use  COMMON;
with CHANNELS;
with CALENDAR;
use  CALENDAR;
with COMLAYER;

procedure MARS is

   IN_MESSAGE      : MESSAGE_FORM;    --input message
   FIRST_MESSAGE   : CONFIG_MESSAGE; --initialization message
   --that contains the chosen routing strategy
   SECOND_MESSAGE : PERIOD_MESSAGE; --initialization message
   --that contains the interval of repetition for the
   --AUV flow execution
   LOCATION        : constant PROGRAMS     :=  MARS; --local
                                            --processor ID
   STOPPER         : constant INTEGER := 100; --number of
   --loop iterations to be run by the AUV flow simulation
   NUMBER_OF_INPUTS: constant NATURAL := 3;   --number of used
   --communication channel inputs
   WHICH_CHANNEL:   INTEGER; -- returned by the call to
   --primitive READ of the package CHANNELS
   --indicating from which channel the message comes in.
   NUMBER_OF_OUTPUTS: constant NATURAL := 3; --number of used
   --communication channel outputs
   --This task receives PILOT_UPDATE message from task
   --VEHICLE_SYS and
   --AP_ORDERS message from task GUIDANCE and sends VS_ORDERS
   --message to task VEHICLE_SYS
   task AUTO_PILOT is
     entry AP_ORDERS    (PILOT_MESSAGE : in MESSAGE_FORM);
     entry PILOT_UPDATE (PILOT_MESSAGE : in MESSAGE_FORM);
   end;
```

```
--This task controls the frequency of execution of the AUV
--flow.It receives the period interval to be used from
--main program MARS during initialization.
task TIMER is
   entry SET_TIMER(SET_PERIOD: in  DURATION);
end;

--This task accepts message GO from from task TIMER and
--message VS_ORDERS from task AUTO_PILOT and sends
--message    SYS_STATUS to task NAVIGATION, message
--UPDATE_SONAR to task SONAR, message TO_MONITOR to task
--MONITOR and message PILOT_UPDATE to task AUTO_PILOT.
--The entry FIN is not used in this implementation.
task VEHICLE_SYS is
   entry VS_ORDERS    (VS_MESSAGE : in MESSAGE_FORM)    ;
   entry GO                                             ;
   entry FIN                                            ;
end;

--communication channels that are used
OutToEarth  : CHANNELS.CHANNEL_REF :=
CHANNELS.OUT_PARAMETERS (2);
InFromEarth : CHANNELS.CHANNEL_REF :=
CHANNELS.IN_PARAMETERS (2);
InFromVenus : CHANNELS.CHANNEL_REF :=
CHANNELS.IN_PARAMETERS (4);
OutToVenus  : CHANNELS.CHANNEL_REF :=
CHANNELS.OUT_PARAMETERS (4);
InFromPluto : CHANNELS.CHANNEL_REF :=
CHANNELS.IN_PARAMETERS (5);
OutToPluto  : CHANNELS.CHANNEL_REF :=
CHANNELS.OUT_PARAMETERS (5);

--Array that contains the input communication channels.
--This array isused by the call to primitive READ of the
--package CHANNELS.
MARS_ARRAY: CHANNELS.CHANNEL_ARRAY(0..NUMBER_OF_INPUTS-1)
:= (InFromEarth, InFromVenus, InFromPluto);

--This array enables each input communication channel
--individually for use with primitive READ of package
--CHANNELS.It is associated with MARS_ARRAY.
MARS_GUARD: CHANNELS.GUARD_ARRAY(0..NUMBER_OF_INPUTS-1) :=
             (TRUE, TRUE, TRUE);--all input channels are
                               --enabled
--Array that contains the output communication channels
--OUT_ARRAY:
CHANNELS.CHANNEL_ARRAY(0..NUMBER_OF_OUTPUTS-1) :=
             (OutToEarth, OutToVenus, OutToPluto);
```

```
      --Table that defines the routing strategy to be used. The
      --type OUT_TABLE is defined in package COMMON.
      MARS_TABLE: OUT_TABLE(EARTH..PLUTO,
      0..NR_OF_OUTPUTS(MARS)-1);

      --This procedure is used for the instantiation of the
      --package COMLAYER. It is used by task QUE in package
      --COMLAYER when sending messages to local tasks AUTO_PILOT
      --and VEHICLE_SYS.
      procedure SEND_IT_FROM_MARS (MESSAGE : in MESSAGE_FORM;
                                          MESS : out BOOLEAN) is

        MESSAGE_SENT : BOOLEAN := FALSE;

      begin
        case MESSAGE.DESTIN is
          when TASK_AUTO_PILOT =>
            case MESSAGE.ENT_CALL is
              when AP_ORDERS =>
                select
                  AUTO_PILOT.AP_ORDERS (MESSAGE);
                  MESSAGE_SENT := TRUE;
                or
                  delay 0.0;
                end select;
              when PILOT_UPDATE =>
                select
                  AUTO_PILOT.PILOT_UPDATE (MESSAGE);
                  MESSAGE_SENT := TRUE;
                or
                  delay 0.0;
                end select;
              when others => null;  --Not a valid call
            end case;
          when TASK_TIMER =>
            case MESSAGE.ENT_CALL is
              when others => null;  --Not a valid call
            end case;
          when TASK_VEHICLE_SYS  =>
            case MESSAGE.ENT_CALL is
              when VS_ORDERS =>
                select
                  VEHICLE_SYS.VS_ORDERS (MESSAGE);
                  MESSAGE_SENT := TRUE;
                or
                  delay 0.0;
                end select;
              when others => null;  --Not a valid call
            end case;
          when others  => null;  --Not a valid task
        end case;
```

108

```
      MESS := MESSAGE_SENT;
      return;
end SEND_IT_FROM_MARS;

--instantiates the package for communication layer.
package MARS_LAYER is new COMLAYER(SEND_IT_FROM_MARS);
use      MARS_LAYER;



task body AUTO_PILOT is
   PILOT_INT                  : constant DURATION := 0.04;
   IN_TASK, NEW_SYS, TALK : MESSAGE_FORM;
   OPEN1: BOOLEAN := TRUE;
   OPEN2: BOOLEAN := TRUE;

begin

   loop
     while (OPEN1 or OPEN2) loop
       select
         when OPEN1 =>
           accept PILOT_UPDATE (PILOT_MESSAGE : in
           MESSAGE_FORM) do
             NEW_SYS := PILOT_MESSAGE;
             OPEN1 := FALSE;
           end PILOT_UPDATE;
       or
         when OPEN2 =>
           accept AP_ORDERS (PILOT_MESSAGE : in
         MESSAGE_FORM) do
             IN_TASK := PILOT_MESSAGE;
             OPEN2 := FALSE;
           end AP_ORDERS;
       or
         terminate;
       end select;
     end loop;

     OPEN1 := TRUE;
     OPEN2 := TRUE;


     delay PILOT_INT;
     IN_TASK.ORIGIN   := TASK_AUTO_PILOT;
     IN_TASK.DESTIN   := TASK_SCREEN;
     IN_TASK.ENT_CALL := OUTPUT;
     IN_TASK.MESSAGE_CODE := 78;
     --INOUT.SEND(IN_TASK); -- debugging message

     IN_TASK.ORIGIN     := TASK_AUTO_PILOT;
```

```
         IN_TASK.DESTIN      := TASK_VEHICLE_SYS;
         IN_TASK.ENT_CALL   := VS_ORDERS;
         INOUT.SEND (IN_TASK);
      end loop;

   end AUTO_PILOT;

   --This task controls the frequency of execution of the
   --AUV flow.It receives the period interval to be used
   --from main program MARS
   --during initialization.
   task body TIMER is

      COUNT   : INTEGER            := 0 ;
      TALK : MESSAGE_FORM;
      NEXT_TIME: CALENDAR.TIME;
      INTERVAL: DURATION;

   begin
      accept SET_TIMER(SET_PERIOD: in DURATION) do
         INTERVAL := SET_PERIOD;
      end SET_TIMER;
      NEXT_TIME := CALENDAR.CLOCK;
      loop
         delay NEXT_TIME - CALENDAR.CLOCK;
         VEHICLE_SYS.GO;
         COUNT := COUNT + 1;
         exit when COUNT = STOPPER;
         NEXT_TIME := NEXT_TIME + INTERVAL;
      end loop;

      delay 3.0;
      TALK                := SHUTDOWN_MESSAGE;
      TALK.DESTIN         := LOOP_TASK;
      TALK.MESSAGE_CODE := 99;
      INOUT.SEND (TALK);

   end TIMER;


   task body VEHICLE_SYS is
      VEHICLE_INT      : constant DURATION := 0.08;
      IN_TASK, TALK    : MESSAGE_FORM;
      NEXT_TIME    : CALENDAR.TIME := CLOCK + VEHICLE_INT;
      PRE_STAMP    : CALENDAR.TIME;
      START_STAMP : CALENDAR.TIME;
      TIMER          : DURATION;
      FINAL          : DURATION;
      COUNT          : INTEGER := 0;

   begin
```

```
      PRE_STAMP := CLOCK;
      TALK.ORIGIN         := TASK_VEHICLE_SYS;
      TALK.DESTIN         := TASK_SCREEN    ;
      TALK.ENT_CALL       := OUTPUT         ;
      TALK.MESSAGE_CODE := 30               ;
      INOUT.SEND (TALK);
      loop
        select
          accept GO;
          START_STAMP := CLOCK;
          delay VEHICLE_INT;

          IN_TASK.ORIGIN := TASK_VEHICLE_SYS;
          IN_TASK.DESTIN := TASK_SCREEN;
          IN_TASK.ENT_CALL := OUTPUT;
          IN_TASK.MESSAGE_CODE := 71;
          --INOUT.SEND (IN_TASK); -- debugging message


          IN_TASK.ORIGIN        := TASK_VEHICLE_SYS;
          IN_TASK.DESTIN        := TASK_NAVIGATION ;
          IN_TASK.ENT_CALL      := SYS_STATUS      ;
          INOUT.SEND (IN_TASK);

          IN_TASK.ORIGIN        := TASK_VEHICLE_SYS;
          IN_TASK.DESTIN        := TASK_SONAR       ;
          IN_TASK.ENT_CALL      := UPDATE_SONAR     ;
          INOUT.SEND (IN_TASK);

          IN_TASK.ORIGIN        := TASK_VEHICLE_SYS;
          IN_TASK.DESTIN        := TASK_MONITOR     ;
          IN_TASK.ENT_CALL      := TO_MONITOR       ;
          INOUT.SEND (IN_TASK);

          IN_TASK.ORIGIN        := TASK_VEHICLE_SYS;
          IN_TASK.DESTIN        := TASK_AUTO_PILOT ;
          IN_TASK.ENT_CALL      := PILOT_UPDATE     ;
          INOUT.SEND (IN_TASK);

          accept VS_ORDERS (VS_MESSAGE : in MESSAGE_FORM) do
            IN_TASK := VS_MESSAGE;
          end VS_ORDERS;


          TIMER := CLOCK - START_STAMP;
          COUNT := COUNT + 1;
          TALK.ORIGIN        := TASK_VEHICLE_SYS;
          TALK.DESTIN        := TASK_SCREEN     ;
          TALK.ENT_CALL      := OUTPUT          ;
          TALK.TIME_STAMP    := TIMER           ;
```

111

```
            TALK.MESSAGE_CODE := 20                    ;
            INOUT.SEND (TALK);

            exit when COUNT = STOPPER;
         or
           accept FIN;
           exit;
         end select;
      end loop;

      FINAL := CLOCK - PRE_STAMP;

      TALK.ORIGIN        := TASK_VEHICLE_SYS;
      TALK.DESTIN        := TASK_SCREEN      ;
      TALK.ENT_CALL      := OUTPUT           ;
      TALK.TIME_STAMP    := FINAL            ;
      TALK.MESSAGE_CODE := 31                ;
      INOUT.SEND (TALK);
   end VEHICLE_SYS;

begin

   --reads the routing strategy initialization message
   CONFIG_IO.READ(InFromEarth, FIRST_MESSAGE);
   FIRST_MESSAGE.MASK(MARS) := TRUE;
   --passes the routing strategy message to PLUTO
   CONFIG_IO.WRITE(OutToPluto, FIRST_MESSAGE);

    --reads the routing strategy message from VENUS and sends
    --it back to EARTH
    CONFIG_IO.READ(InFromVenus, FIRST_MESSAGE);
    CONFIG_IO.WRITE(OutToEarth, FIRST_MESSAGE);

    --reads the period interval initialization message
    PERIOD_IO.READ(InFromEarth, SECOND_MESSAGE);
    SECOND_MESSAGE.MASK(MARS) := TRUE;

    --passes the period interval message to PLUTO
    PERIOD_IO.WRITE(OutToPluto, SECOND_MESSAGE);

    --reads the period interval message from VENUS and sends
    --it back to EARTH
    PERIOD_IO.READ(InFromVenus, SECOND_MESSAGE);
    PERIOD_IO.WRITE(OutToEarth, SECOND_MESSAGE);

    --initialize the routing strategy table
    MARS_TABLE := MARS_CONFIG(FIRST_MESSAGE.ROUT_INFO);

    --set the timer with the period interval
    TIMER.SET_TIMER(SECOND_MESSAGE.PERIOD_INFO);
```

```
      --initialization of task INOUT
     INOUT.INIT1(LOCATION);    --main node program ID
     INOUT.INIT2(OUT_ARRAY);   --output channel array
     INOUT.INIT3(MARS_TABLE); --routing table


   loop
     --Reads message from a input channel
      MESSAGE_IO.READ(MARS_ARRAY, MARS_GUARD, WHICH_CHANNEL,
    IN_MESSAGE);

     --Increments MARS counter in the PROG(2) message field
     IN_MESSAGE.PROG(2) := IN_MESSAGE.PROG(2) + 1;

     --if a shutdown message has arrived exit the loop
     if IN_MESSAGE.ORIGIN = SHUTDOWN  and IN_MESSAGE.DESTIN
      = HOST_TASK then
        IN_MESSAGE.PROG(2) := -1 * IN_MESSAGE.PROG(2);
        delay 1.0;
        INOUT.INCOMING (IN_MESSAGE);
        exit;
     end if;

     --sends input message to the traffic handler
     INOUT.INCOMING (IN_MESSAGE);
   end loop;

end MARS;


       2.   PLUTO.ADA


with COMMON;
use  COMMON;
with CHANNELS;
with CALENDAR;
use  CALENDAR;
with COMLAYER;

procedure PLUTO is

   use COMMON;
   use CALENDAR;

   IN_MESSAGE      : MESSAGE_FORM;    --input message

   FIRST_MESSAGE   : CONFIG_MESSAGE; --initialization message
   --that contains the chosen routing strategy
```

```
SECOND_MESSAGE : PERIOD_MESSAGE; --initialization message
--that contains the interval of repetition of the
--AUV flow execution

LOCATION          : constant PROGRAMS      :=  PLUTO; --local
--processor ID

NUMBER_OF_INPUTS: constant NATURAL      := 2; --number of
--used communication channel inputs

WHICH_CHANNEL    : INTEGER; --returned by the call to
--primitive READ of package CHANNELS indicating from
--which channel the message comes in.
NUMBER_OF_OUTPUTS: constant NATURAL := 2; --number of used
--communication channel outputs
--This task receives message MONITOR_UPDATE from task
--MONITOR and message OBJECT_ALERT from task AVOIDANCE. It
--sends  message UPDATE_ORDERS to task GUIDANCE.
task EXE_MISSION is
  entry OBJECT_ALERT (EXE_MESSAGE : in MESSAGE_FORM);
  entry MONITOR_UPDATE (EXE_MESSAGE : in MESSAGE_FORM);
end;

--This task receives message TO_MONITOR from task
--VEHICLE_SYS and sends message MONITOR_UPDATE to task
--EXE_MISSION
task MONITOR is
  entry TO_MONITOR (MON_MESSAGE : in MESSAGE_FORM);
end;

-- communication channels that are used
OutToSaturn : CHANNELS.CHANNEL_REF :=
CHANNELS.OUT_PARAMETERS (4);
InFromSaturn: CHANNELS.CHANNEL_REF :=
CHANNELS.IN_PARAMETERS (4);
InFromMars   : CHANNELS.CHANNEL_REF :=
CHANNELS.IN_PARAMETERS (3);
OutToMars    : CHANNELS.CHANNEL_REF :=
CHANNELS.OUT_PARAMETERS(3);

--Array that contains the input communication channels.
--This array is used by the call to primitive READ of
--package CHANNELS.
PLUTO_ARRAY: CHANNELS.CHANNEL_ARRAY(0..NUMBER_OF_INPUTS-1)
:= (InFromSaturn, InFromMars);

--This array enables each input communication channel
--individually
--for use with primitive READ of package CHANNELS.
--It is associated with PLUTO_ARRAY.
```

```ada
PLUTO_GUARD: CHANNELS.GUARD_ARRAY(0..NUMBER_OF_INPUTS-1)
:= (TRUE, TRUE);

--Array that contains the output communication channels
--OUT_ARRAY:
CHANNELS.CHANNEL_ARRAY(0..NUMBER_OF_OUTPUTS-1) :=
              (OutToMars, OutToSaturn);

--Table that defines the routing strategy to be used. The
--type
--OUT_TABLE is defined in package COMMON.
PLUTO_TABLE:
OUT_TABLE(EARTH..PLUTO,0..NR_OF_OUTPUTS(PLUTO)-1);

--This procedure is used for the instantiation of the
--package COMLAYER. It is used by task QUE in package
--COMLAYER when sending messages to local tasks.
procedure SEND_IT_FROM_PLUTO (MESSAGE : in MESSAGE_FORM;
                              MESS : out BOOLEAN) is

  MESSAGE_SENT : BOOLEAN := FALSE;
begin
  case MESSAGE.DESTIN is
    when TASK_EXE_MISSION  =>
      case MESSAGE.ENT_CALL is
        when OBJECT_ALERT =>
          select
            EXE_MISSION.OBJECT_ALERT(MESSAGE);
            MESSAGE_SENT := TRUE;
          or
            delay 0.0;
          end select;
        when MONITOR_UPDATE =>
          select
            EXE_MISSION.MONITOR_UPDATE(MESSAGE);
            MESSAGE_SENT := TRUE;
          or
            delay 0.0;
          end select;
        when others => null;  -- Not a valid call
      end case;
    when TASK_MONITOR  =>
      case MESSAGE.ENT_CALL is
        when TO_MONITOR =>
          select
            MONITOR.TO_MONITOR (MESSAGE);
            MESSAGE_SENT := TRUE;
          or
            delay 0.0;
          end select;
        when others => null;  -- Not a valid call
```

```
            end case;
        when others  => null;  -- not a valid task
      end case;

   MESS := MESSAGE_SENT;
   return;
end SEND_IT_FROM_PLUTO;

-- instantiates the package for communication layer
package PLUTO_LAYER is new COMLAYER(SEND_IT_FROM_PLUTO);
use PLUTO_LAYER;


task body EXE_MISSION is

   EXE_INT: constant DURATION := 0.06;
   TALK, IN_TASK : MESSAGE_FORM;
   OPEN1: BOOLEAN := TRUE;
   OPEN2: BOOLEAN := TRUE;

begin

   loop
     while (OPEN1 or OPEN2) loop
       select
         when OPEN1 =>
           accept MONITOR_UPDATE (EXE_MESSAGE : in
           MESSAGE_FORM) do
             IN_TASK := EXE_MESSAGE;
             OPEN1 := FALSE;
           end MONITOR_UPDATE;
       or
           when OPEN2 =>
             accept OBJECT_ALERT (EXE_MESSAGE : in
             MESSAGE_FORM) do
               IN_TASK := EXE_MESSAGE;
               OPEN2 := FALSE;
             end OBJECT_ALERT;
       or
          terminate;
        end select;
     end loop;

     OPEN1 := TRUE;
     OPEN2 := TRUE;
     delay EXE_INT;

     IN_TASK.ORIGIN := TASK_EXE_MISSION;
     IN_TASK.DESTIN := TASK_SCREEN;
     IN_TASK.ENT_CALL := OUTPUT;
     IN_TASK.MESSAGE_CODE := 76;
```

116

```
      --INOUT.SEND (IN_TASK); debugging message

      IN_TASK.ORIGIN      := TASK_EXE_MISSION;
      IN_TASK.DESTIN      := TASK_GUIDANCE;
      IN_TASK.ENT_CALL    := UPDATE_ORDERS;
      INOUT.SEND (IN_TASK);

   end loop;
end EXE_MISSION;

--This task receives message TO_MONITOR from task
--VEHICLE_SYS and sends message MONITOR_UPDATE to task
--EXE_MISSION
task body MONITOR is
   MONITOR_INT: constant DURATION := 0.03;
   IN_TASK, TALK : MESSAGE_FORM;
begin

   loop
     select
       accept TO_MONITOR (MON_MESSAGE : in MESSAGE_FORM)
       do
          IN_TASK := MON_MESSAGE;
       end TO_MONITOR;
       delay MONITOR_INT;


       IN_TASK.ORIGIN := TASK_MONITOR;
       IN_TASK.DESTIN := TASK_SCREEN;
       IN_TASK.ENT_CALL := OUTPUT;
       IN_TASK.MESSAGE_CODE := 74;
       -- INOUT.SEND( IN_TASK); -- debugging message


       IN_TASK.DESTIN      := TASK_EXE_MISSION;
       IN_TASK.ORIGIN      := TASK_MONITOR;
       IN_TASK.ENT_CALL    := MONITOR_UPDATE;


       INOUT.SEND (IN_TASK);
     or
       terminate;
     end select;
   end loop;

end MONITOR;

begin

   --reads the routing strategy initialization message
   CONFIG_IO.READ(InFromMars, FIRST_MESSAGE);
```

117

```
      FIRST_MESSAGE.MASK(PLUTO) := TRUE;

      --passes the routing strategy message to SATURN
      CONFIG_IO.WRITE(OutToSaturn, FIRST_MESSAGE);

      --reads the period interval initialization message
      PERIOD_IO.READ(InFromMars, SECOND_MESSAGE);
      SECOND_MESSAGE.MASK(PLUTO) := TRUE;

      --passes the period interval message to SATURN
      PERIOD_IO.WRITE(OutToSaturn, SECOND_MESSAGE);

      --initialize the routing strategy table
      PLUTO_TABLE := PLUTO_CONFIG(FIRST_MESSAGE.ROUT_INFO);

      --initialization of task INOUT
      INOUT.INIT1(LOCATION);      -- main node program ID
      INOUT.INIT2(OUT_ARRAY);     -- output channel array
      INOUT.INIT3(PLUTO_TABLE); -- routing table

      loop
        --Reads message from a input channel
        MESSAGE_IO.READ(PLUTO_ARRAY, PLUTO_GUARD, WHICH_CHANNEL,
                        IN_MESSAGE);

        --increments PLUTO counter in PROG(5) message field
        IN_MESSAGE.PROG(5) := IN_MESSAGE.PROG(5) + 1;

        --if a shutdown message has arrived exit the loop
        if IN_MESSAGE.ORIGIN = SHUTDOWN then
          IN_MESSAGE.DESTIN   := HOST_TASK;
          IN_MESSAGE.PROG(5)  := -1 * IN_MESSAGE.PROG(5);
          INOUT.INCOMING (IN_MESSAGE);
          exit;
        end if;

        --sends input message to the traffic handler(task INOUT)
        INOUT.INCOMING (IN_MESSAGE);
      end loop;
end PLUTO;


    3.  SATURN.ADA

with COMMON;
use  COMMON;
with CHANNELS;
with CALENDAR;
use  CALENDAR;
with COMLAYER;
```

118

```
procedure SATURN is

   IN_MESSAGE       : MESSAGE_FORM;   -- inputmessage
   FIRST_MESSAGE    : CONFIG_MESSAGE; --initialization message
   --that contains the chosen routing strategy
   SECOND_MESSAGE : PERIOD_MESSAGE; --initialization message
   --that contains the interval of repetition of the
   --AUV flow execution.
   LOCATION         : constant PROGRAMS  :=  SATURN; --local
   --processor ID

   NUMBER_OF_INPUTS: constant NATURAL   :=  2; -- number of
   --used communication channels inputs
   WHICH_CHANNEL: INTEGER; -- returned by the call to
   --primitive READ of package CHANNELS indicating from
   --which channel the message comes in.
   NUMBER_OF_OUTPUTS : constant NATURAL  := 2; --number of
--used communication channels output.

   task AVOIDANCE is
     entry OB_AVOID (AVOID_MESSAGE : in MESSAGE_FORM);
   end;


   task GUIDANCE is
     entry UPDATE_NAV     (GUIDE_MESSAGE : in MESSAGE_FORM);
     entry UPDATE_ORDERS (GUIDE_MESSAGE : in MESSAGE_FORM);
     entry AVOID_REC      (GUIDE_MESSAGE : in MESSAGE_FORM);
   end;

   -- communication channels that are used
   OutToVenus  : CHANNELS.CHANNEL_REF :=
CHANNELS.OUT_PARAMETERS (3);
   InFromVenus : CHANNELS.CHANNEL_REF :=
CHANNELS.IN_PARAMETERS (3);
   InFromPluto : CHANNELS.CHANNEL_REF :=
CHANNELS.IN_PARAMETERS (2);
   OutToPluto  : CHANNELS.CHANNEL_REF :=
CHANNELS.OUT_PARAMETERS (2);

   --Array that contains the input communication channels.
--This array is used by the call to primitive READ of
--package CHANNELS.
   SATURN_ARRAY:
CHANNELS.CHANNEL_ARRAY(0..NUMBER_OF_INPUTS-1)  :=
                 (InFromVenus, InFromPluto);

   --This array enables each input communication channel
--individually
   --for use with primitive READ of package CHANNELS.
   --It is associated with SATURN_ARRAY.
```

```ada
     SATURN_GUARD: CHANNELS.GUARD_ARRAY(0..NUMBER_OF_INPUTS-1)
   := (TRUE, TRUE);

   --Array that contains the output communication channels
   OUT_ARRAY:
CHANNELS.CHANNEL_ARRAY(0..NUMBER_OF_OUTPUTS-1)
   := (OutToVenus, OutToPluto);

   --Table that defines the routing strategy to be used. The
  --type OUT_TABLE is defined in package COMMON
   SATURN_TABLE: OUT_TABLE(EARTH..PLUTO,
0..NR_OF_OUTPUTS(SATURN)-1);

   --This procedure is used for the instantiation of the
--package COMLAYER. It is used by task QUE in package
--COMLAYER when sending messages to local tasks.
   procedure SEND_IT_FROM_SATURN (MESSAGE : in MESSAGE_FORM;
                                   MESS : out BOOLEAN) is

     MESSAGE_SENT : BOOLEAN := FALSE;

   begin
     case MESSAGE.DESTIN is
       when TASK_AVOIDANCE =>
         case MESSAGE.ENT_CALL is
           when OB_AVOID =>
             select
               AVOIDANCE.OB_AVOID (MESSAGE);
               MESSAGE_SENT := TRUE;
             or
               delay 0.0;
             end select;
           when others => null; -- not a valid call
         end case;
       when TASK_GUIDANCE =>
         case MESSAGE.ENT_CALL is
           when UPDATE_NAV =>
             select
               GUIDANCE.UPDATE_NAV (MESSAGE);
               MESSAGE_SENT := TRUE;
             or
               delay 0.0;
             end select;
           when UPDATE_ORDERS =>
             select
               GUIDANCE.UPDATE_ORDERS (MESSAGE);
               MESSAGE_SENT := TRUE;
             or
               delay 0.0;
             end select;
           when AVOID_REC =>
```

120

```
          select
            GUIDANCE.AVOID_REC (MESSAGE);
            MESSAGE_SENT := TRUE;
          or
            delay 0.0;
          end select;
        when others => null;  -- Not a valid call
      end case;
    when others  => null;  -- not a valid task
  end case;

  MESS := MESSAGE_SENT;
  return;
end SEND_IT_FROM_SATURN;

--instantiates the package for communication layer
package SATURN_LAYER is new COMLAYER(SEND_IT_FROM_SATURN);
use SATURN_LAYER;


task body AVOIDANCE is

  AVOIDANCE_INT: constant DURATION := 0.08;
  TALK, IN_TASK : MESSAGE_FORM;

begin

  loop
    select
      accept OB_AVOID (AVOID_MESSAGE : in MESSAGE_FORM) do
        IN_TASK := AVOID_MESSAGE;
      end OB_AVOID;

      delay AVOIDANCE_INT;

      IN_TASK.ORIGIN     := TASK_AVOIDANCE;
      IN_TASK.DESTIN     := TASK_SCREEN;
      IN_TASK.ENT_CALL   := OUTPUT;
      IN_TASK.MESSAGE_CODE := 75;
      -- INOUT.SEND( IN_TASK); -- debugging message

      IN_TASK.ORIGIN     := TASK_AVOIDANCE ;
      IN_TASK.DESTIN     := TASK_GUIDANCE  ;
      IN_TASK.ENT_CALL   := AVOID_REC      ;
      INOUT.SEND (IN_TASK);

      IN_TASK.ORIGIN     := TASK_AVOIDANCE  ;
      IN_TASK.DESTIN     := TASK_EXE_MISSION;
      IN_TASK.ENT_CALL   := OBJECT_ALERT    ;
      INOUT.SEND (IN_TASK);
```

```
      or
         terminate;
      end select;
   end loop;

end AVOIDANCE;


task body GUIDANCE is

   GUIDANCE_INT: constant DURATION := 0.07;
   EMERG, IN_TASK, GO_TO, WE_ARE, TALK : MESSAGE_FORM;
   OPEN1: BOOLEAN := TRUE;
   OPEN2: BOOLEAN := TRUE;
   OPEN3: BOOLEAN := TRUE;


begin

   loop

      while (OPEN1 or OPEN2 or OPEN3) loop
         select
            when OPEN1 =>
               accept UPDATE_NAV (GUIDE_MESSAGE : in
               MESSAGE_FORM) do
                  WE_ARE := GUIDE_MESSAGE;
                  OPEN1 := FALSE;
               end UPDATE_NAV;
         or
            when OPEN2 =>
               accept AVOID_REC (GUIDE_MESSAGE : in
               MESSAGE_FORM) do
                  GO_TO := GUIDE_MESSAGE;
                  OPEN2 := FALSE;
               end AVOID_REC;

         or
            when OPEN3 =>
               accept UPDATE_ORDERS (GUIDE_MESSAGE : in
               MESSAGE_FORM) do
                  GO_TO := GUIDE_MESSAGE;
                  OPEN3 := FALSE;
               end UPDATE_ORDERS;
         or
            terminate;
         end select;

      end loop;

      OPEN1 := TRUE;
```

122

```
         OPEN2 := TRUE;
         OPEN3 := TRUE;

         delay GUIDANCE_INT;

         IN_TASK.ORIGIN       := TASK_GUIDANCE;
         IN_TASK.DESTIN       := TASK_SCREEN;
         IN_TASK.ENT_CALL     := OUTPUT;
         IN_TASK.MESSAGE_CODE := 77;
         --INOUT.SEND(IN_TASK); -- debugging message

         IN_TASK.ORIGIN       := TASK_GUIDANCE   ;
         IN_TASK.DESTIN       := TASK_AUTO_PILOT;
         IN_TASK.ENT_CALL     := AP_ORDERS        ;
         INOUT.SEND (IN_TASK);
      end loop;

   end GUIDANCE;

begin

   --reads the routing strategy initialization message
   CONFIG_IO.READ(InFromPluto, FIRST_MESSAGE);
   FIRST_MESSAGE.MASK(SATURN) := TRUE;

   --passes the routing strategy to VENUS
   CONFIG_IO.WRITE(OutToVenus, FIRST_MESSAGE);

   --reads the period interval initialization message
   PERIOD_IO.READ(InFromPluto, SECOND_MESSAGE);
   SECOND_MESSAGE.MASK(SATURN) := TRUE;

   --passes the period interval message to SATURN.
   PERIOD_IO.WRITE(OutToVenus, SECOND_MESSAGE);

   --initialize the routing strategy table
   SATURN_TABLE := SATURN_CONFIG(FIRST_MESSAGE.ROUT_INFO);

   --initialization of task INOUT
   INOUT.INIT1(LOCATION);       -- main node program ID
   INOUT.INIT2(OUT_ARRAY);      -- output channel array
   INOUT.INIT3(SATURN_TABLE); -- routing table

   loop
      --Reads message from a input channel
      MESSAGE_IO.READ (SATURN_ARRAY, SATURN_GUARD,
   WHICH_CHANNEL,IN_MESSAGE);

      --increments SATURN counter in PROG(4) message field
      IN_MESSAGE.PROG(4) := IN_MESSAGE.PROG(4) + 1;
```

```
      --if a shutdown message has arrived exit the loop
      if IN_MESSAGE.ORIGIN = SHUTDOWN  and IN_MESSAGE.DESTIN =
        HOST_TASK then
        IN_MESSAGE.PROG(4) := -1 * IN_MESSAGE.PROG(4);
        INOUT.INCOMING (IN_MESSAGE);
        exit;
      end if;

      --sends input message to the traffic handler(task INOUT)
      INOUT.INCOMING (IN_MESSAGE);
    end loop;
end SATURN;
```

4.   VENUS.ADA

```
with COMMON;
use  COMMON;
with CHANNELS;
with CALENDAR;
use  CALENDAR;
with COMLAYER;

procedure VENUS is

  IN_MESSAGE      : MESSAGE_FORM; --input message
  FIRST_MESSAGE   : CONFIG_MESSAGE; -- initialization message
  --that contains the chosen routing strategy
  SECOND_MESSAGE : PERIOD_MESSAGE; -- initialization message
  --that contains the interval of repetition of the
  --AUV flow execution
  LOCATION         : constant PROGRAMS     :=  VENUS; --local
  --processor ID
  NUMBER_OF_INPUTS: constant NATURAL  :=  2; --number of
  --used communication channel inputs
  WHICH_CHANNEL:  INTEGER;  --returned by the call to
  --primitive READ of package CHANNELS indicating from
  --which channel the message comes in.
  NUMBER_OF_OUTPUTS: constant NATURAL   :=  2; -- number of
  --used communication channel outputs

  task NAVIGATION is
    entry SONAR_OBSTICLE (NAV_MESSAGE : in MESSAGE_FORM);
    entry SYS_STATUS     (NAV_MESSAGE : in MESSAGE_FORM);
  end;


  task SONAR is
    entry UPDATE_SONAR (SONAR_MESSAGE : in MESSAGE_FORM) ;
  end;
```

124

```
--communication channels that are used
InFromMars    : CHANNELS.CHANNEL_REF :=
CHANNELS.IN_PARAMETERS (2);
OutToMars     : CHANNELS.CHANNEL_REF :=
CHANNELS.OUT_PARAMETERS (2);
InFromSaturn : CHANNELS.CHANNEL_REF :=
CHANNELS.IN_PARAMETERS (5);
OutToSaturn   : CHANNELS.CHANNEL_REF :=
CHANNELS.OUT_PARAMETERS (5);

--Array that contains the input communication channels.
--This array is used by the call to primitive READ of the
--package CHANNELS.
VENUS_ARRAY:
CHANNELS.CHANNEL_ARRAY(0..NUMBER_OF_INPUTS-1) :=
(InFromMars, InFromSaturn);
--This array enables each input communication channel
--individually for use with primitive READ of package
--CHANNELS.It is associated with VENUS_ARRAY.
VENUS_GUARD: CHANNELS.GUARD_ARRAY(0..NUMBER_OF_INPUTS-1)
:= (TRUE, TRUE);

-- Array that contains the output communication channels
OUT_ARRAY: CHANNELS.CHANNEL_ARRAY(0..NUMBER_OF_OUTPUTS-1)
:= (OutToMars, OutToSaturn);

--Table that defines the routing strategy to be used. The
--type OUT_TABLE is defined in package COMMON
VENUS_TABLE: OUT_TABLE(EARTH..PLUTO,
0..NR_OF_OUTPUTS(VENUS)-1);

--This procedure is used for the instantiation of the
--package COMLAYER. It is used by task QUE in package
--COMLAYER when sending messages to local tasks.
procedure SEND_IT_FROM_VENUS (MESSAGE : in MESSAGE_FORM;
                              MESS : out BOOLEAN) is

   MESSAGE_SENT : BOOLEAN := FALSE;
begin
   case MESSAGE.DESTIN is
     when TASK_NAVIGATION  =>
       case MESSAGE.ENT_CALL is
         when SONAR_OBSTICLE =>
           select
             NAVIGATION.SONAR_OBSTICLE(MESSAGE);
             MESSAGE_SENT := TRUE;
           or
             delay 0.0;
           end select;
         when SYS_STATUS =>
           select
```

```
                    NAVIGATION.SYS_STATUS(MESSAGE);
                    MESSAGE_SENT := TRUE;
                or
                    delay 0.0;
                end select;
            when others => null;   -- Not a valid call
          end case;
     when TASK_SONAR =>
        case MESSAGE.ENT_CALL is
          when UPDATE_SONAR =>
            select
                SONAR.UPDATE_SONAR (MESSAGE);
                MESSAGE_SENT := TRUE;
            or
                delay 0.0;
            end select;
            when others => null;   -- Not a valid call
          end case;
      when others  => null;   -- not a valid task
   end case;

  MESS := MESSAGE_SENT;
  return;
end SEND_IT_FROM_VENUS;

--instantiates the package for communication layer
package VENUS_LAYER is new COMLAYER(SEND_IT_FROM_VENUS);
use VENUS_LAYER;

--This task receives SYS_STATUS message  from task
--VEHICLE_SYS and SONAR_OBSTICLE message from task SONAR.
--It sends UPDATE_NAV message to task GUIDANCE.
task body  NAVIGATION is

  NAVIGATION_INT: constant DURATION := 0.05;
  SYNC_INT      : DURATION := 0.03;
  IN_TASK, TALK : MESSAGE_FORM;
  OPEN1: BOOLEAN := TRUE;
  OPEN2: BOOLEAN := TRUE;

begin

  loop
    while (OPEN1 or OPEN2) loop
      select
        when OPEN1 =>
          accept SYS_STATUS (NAV_MESSAGE : in
        MESSAGE_FORM) do
            IN_TASK := NAV_MESSAGE;
            OPEN1 := FALSE;
          end SYS_STATUS;
```

```
      or
        when OPEN2 =>
          accept SONAR_OBSTICLE (NAV_MESSAGE : in
        MESSAGE_FORM) do
            IN_TASK := NAV_MESSAGE;
            OPEN2 := FALSE;
          end SONAR_OBSTICLE;
      or
         terminate;
      end select;
    end loop;


    OPEN1 := TRUE;
    OPEN2 := TRUE;

    delay NAVIGATION_INT;

    IN_TASK.ORIGIN      := TASK_NAVIGATION;
    IN_TASK.DESTIN      := TASK_SCREEN;
    IN_TASK.ENT_CALL    := OUTPUT;
    IN_TASK.MESSAGE_CODE := 73;
    --INOUT.SEND( IN_TASK ); -- debugging message

    IN_TASK.ORIGIN       := TASK_NAVIGATION;
    IN_TASK.DESTIN       := TASK_GUIDANCE;
    IN_TASK.ENT_CALL     := UPDATE_NAV;
    INOUT.SEND (IN_TASK);
  end loop;

end NAVIGATION;


task body SONAR is

  SONAR_INT: constant DURATION := 0.02;
  TALK, IN_TASK          : MESSAGE_FORM ;
  EMERG_MESSAGE   : MESSAGE_FORM ;

begin

  loop
    select
      accept UPDATE_SONAR (SONAR_MESSAGE : in
      MESSAGE_FORM) do
        IN_TASK := SONAR_MESSAGE;
      end UPDATE_SONAR;


      delay SONAR_INT;
```

```
            IN_TASK.ORIGIN   := TASK_SONAR;
            IN_TASK.DESTIN   := TASK_SCREEN;
            IN_TASK.ENT_CALL := OUTPUT;
            IN_TASK.MESSAGE_CODE := 72;
            --INOUT.SEND( IN_TASK ); -- debugging message

            EMERG_MESSAGE.ORIGIN    := TASK_SONAR     ;
            EMERG_MESSAGE.DESTIN    := TASK_AVOIDANCE;
            EMERG_MESSAGE.ENT_CALL  := OB_AVOID       ;
            INOUT.SEND (EMERG_MESSAGE)                ;


            IN_TASK.ORIGIN    := TASK_SONAR     ;
            IN_TASK.DESTIN    := TASK_NAVIGATION;
            IN_TASK.ENT_CALL  := SONAR_OBSTICLE ;
            INOUT.SEND (IN_TASK);
        or
          terminate;
        end select;
      end loop;

    end SONAR;

  begin

    --reads the routing strategy initialization message
    CONFIG_IO.READ(InFromSaturn, FIRST_MESSAGE);
    FIRST_MESSAGE.MASK(VENUS) := TRUE;
    --passes the routing strategy message to MARS
    CONFIG_IO.WRITE(OutToMars, FIRST_MESSAGE);

    --reads the period interval initialization message
    PERIOD_IO.READ(InFromSaturn, SECOND_MESSAGE);
    SECOND_MESSAGE.MASK(VENUS) := TRUE;

    --passes the period interval message to MARS
    PERIOD_IO.WRITE(OutToMars, SECOND_MESSAGE);

    --initialize the routing strategy table
    VENUS_TABLE := VENUS_CONFIG(FIRST_MESSAGE.ROUT_INFO);

    --initialization of task INOUT
    INOUT.INIT1(LOCATION);     -- main node program ID
    INOUT.INIT2(OUT_ARRAY);    -- output channel array
    INOUT.INIT3(VENUS_TABLE);  -- routing table

    loop
      --Reads message from a input channel
      MESSAGE_IO.READ (VENUS_ARRAY, VENUS_GUARD,
    WHICH_CHANNEL,IN_MESSAGE);
```

```
     --increments VENUS counter in PROG(3) message field
     IN_MESSAGE.PROG(3) := IN_MESSAGE.PROG(3) + 1;

     --if a shutdown message has arrived exit the loop
     if IN_MESSAGE.ORIGIN = SHUTDOWN  and IN_MESSAGE.DESTIN =
     HOST_TASK then
        IN_MESSAGE.PROG(3) := -1 * IN_MESSAGE.PROG(3);
        INOUT.INCOMING (IN_MESSAGE);
        exit;
     end if;

     --sends input message to the traffic handler(task INOUT)
     INOUT.INCOMING (IN_MESSAGE);
  end loop;

end VENUS;
```

APPENDIX C: BASIC ADA PACKAGES USED IN TASK ALLOCATION

A.   DISET.ADA

```
generic
  type ATOM is range <>; -- Must be a discrete type

package DISCRETE_SET is

  type SET is private;
  type ATOM_LIST is array (INTEGER range <>) of ATOM;
  function CREATE return SET;
  function UNION (A, B: SET) return SET;
  function INTERSECTION (A, B : SET) return SET;
  function DIFFERENCE (A, B: SET) return SET;
  function COPY (A: SET) return SET;
  function BUILD_SET (L: ATOM_LIST) return SET;
  procedure INSERT (A: ATOM; S: in out SET);
  procedure DELETE (A: ATOM; S: in out SET);
  function MEMBER(A: ATOM; S: SET) return BOOLEAN;
  function SUBSET(A, B: SET) return BOOLEAN;
  function EQUAL(A,B: SET) return BOOLEAN;
  function IS_FULL(A: SET) return BOOLEAN;
  procedure TAKE_OUT_MEMBER(A: in out SET;
                            OUT_ATOM: out ATOM;
                            SUCCESS: out BOOLEAN);
  function COUNT_MEMBERS(A: SET) return INTEGER;
  procedure CLEAR_SET(S: in out SET);

private
  type SET is array (ATOM'FIRST..ATOM'LAST) of BOOLEAN;

end DISCRETE_SET;


package body DISCRETE_SET is

  function T return BOOLEAN renames TRUE;
  function F return BOOLEAN renames FALSE;

  function CREATE return SET is
  begin
    return(others => F);
```

```
end CREATE;

function UNION (A,B : SET) return SET is
begin
  return A or B;
end UNION;

function INTERSECTION (A,B: SET) return SET is
begin
  return A and B;
end INTERSECTION;

function DIFFERENCE (A,B : SET) return SET is
begin
  return A and (A xor B);
end DIFFERENCE;

function COPY (A: SET) return SET is
begin
  return A;
end COPY;

function BUILD_SET (L: ATOM_LIST) return SET is

  S: SET := CREATE;
begin
  for I in L'RANGE loop
    INSERT(L(I),S);
  end loop;

  return S;
end BUILD_SET;

procedure INSERT (A: ATOM; S: in out SET) is

begin
  S(A) := T;
end INSERT;

procedure DELETE (A: ATOM; S: in out SET) is

begin
  S(A) := F;
end DELETE;

function MEMBER (A: ATOM; S: SET) return BOOLEAN is
begin
  return S(A);
end MEMBER;

function SUBSET (A,B : SET) return BOOLEAN is
```

131

```
begin
   return (A and B) = B;
end SUBSET;

function EQUAL (A,B: SET) return BOOLEAN is
begin
   return A=B;
end EQUAL;


function IS_FULL(A: SET) return BOOLEAN is

begin
   for I in A'range loop
     if A(I) = F then
        return F;
     end if;
   end loop;
   return T;
end IS_FULL;


procedure TAKE_OUT_MEMBER(A: in out SET;
                          OUT_ATOM: out ATOM;
                          SUCCESS: out BOOLEAN) is

begin
   SUCCESS := F;
   for I in A'range loop
     if A(I) = T then
        OUT_ATOM := I;
        SUCCESS := T;
        A(I) := F;
        exit;
     else
        OUT_ATOM := I;
     end if;
   end loop;
end TAKE_OUT_MEMBER;

function COUNT_MEMBERS(A: SET) return INTEGER is

   COUNT: INTEGER := 0;

begin
   for I in A'range loop
     if A(I) = T then
        COUNT := COUNT + 1;
     end if;
   end loop;
   return COUNT;
```

```
   end COUNT_MEMBERS;

   procedure CLEAR_SET(S: in out SET) is

      SUCCESS: BOOLEAN := TRUE;
      A: ATOM;

   begin
      while SUCCESS loop
        TAKE_OUT_MENBER(S,A,SUCCESS);
      end loop;
   end CLEAR_SET;


end DISCRETE_SET;
```

B. GRAPH2.ADA

```
generic

   --Types to be instantiated
   type STANDARD_ELEMENT is private;
   type KEY_TYPE is private;

   --Function to be instantiated
   with function ">"(K1, K2: KEY_TYPE) return BOOLEAN;

   -- Procedures to be instantiated
   with procedure PROCESS(E: STANDARD_ELEMENT; EKEY:
KEY_TYPE);


package GRAPH2_ADT is

   --Elements: A graph consists of nodes and edges. Although,
   --in general each node may contain many elements, this
   --package defines that each node will contain exactly one
   --element of type STANDARD_ELEMENT. Since an element
   --is assumed to have a unique key, each node is uniquely
   --identified by the key value of the element it contains.
   --The key value has a type KEY_TYPE.

   --Structure: An edge is a one-to-one relationship between
   --a pair of distinct  nodes. A pair of nodes can be
   --connected by at most one edge, but any node
   --can be connected to any collection of other nodes.

   --Domain: The number of elements in the graph is bounded.
```

133

```
type DGRAPH is private;
DGRAPH_ERROR: exception;

--Operations: If G is a graph then reference to G-pre in a
--postcondition is a reference to the value of G just
--prior to the operation.

procedure INSERT_NODE(G: in out DGRAPH;
                      TKEY: in KEY_TYPE;
                      E: in STANDARD_ELEMENT;
                      SUCCESS: out BOOLEAN);
--pre - None.
--post - If G-pre is not full and does not contain an
--element whose key value is e.key , then G contains e
--and INSERT_NODE is true, else INSERT_NODE is false.

procedure INSERT_EDGE(G: in out DGRAPH;
                      FROM_KEY, TO_KEY: in KEY_TYPE;
                      W: in STANDARD_ELEMENT;
                      SUCCESS: out BOOLEAN);

--pre - None.
--post - If KEY1 /= KEY2, and G-pre is not full,
--contains nodes n1 and n2 with elements whose key
--values are KEY1 and KEY2, and does not contain an edge
--connecting those nodes, then G contains an edge
--connecting n1 and n2 and INSERT_EDGE is true, else
--INSERT_NODE is false.


procedure RETRIEVE_EDGE(G: in out DGRAPH;
                        FROM_KEY, TO_KEY: in KEY_TYPE;
                        W: out STANDARD_ELEMENT;
                        SUCCESS: out BOOLEAN);

--pre - None.
--post - Returns the weight and success = true if the
--edge exists if the edge does not exist returns
--success = false.

procedure DELETE_NODE(G: in out DGRAPH;
                      TKEY: in KEY_TYPE;
                      SUCCESS: out BOOLEAN);

--pre - None.
--post - If G-pre contains node n1, whose element has key
--value KEY1, then G does not contain n1 or any of the
--edges that connected n1 to other nodes  in G-pre, and
--DELETE_NODE is true, else DELETE_NODE is false.
```

```
procedure DELETE_EDGE(G: in out DGRAPH;
                      TKFROM, TKTO: in KEY_TYPE;
                      SUCCESS: out BOOLEAN);

--pre - None.
--post - If G-pre contains an edge connecting nodes whose
--elements have key values KEY1 and KEY2 then G does not
--contain that edge and DELETE_EDGE is true, else
--DELETE_EDGE is false.

procedure UPDATE_NODE(G: in out DGRAPH;
                      TKEY: in KEY_TYPE;
                      E: in STANDARD_ELEMENT;
                      SUCCESS: out BOOLEAN);

--pre - None.
--post - If G-pre contains element e-pre with key value
--e.KEY1 then G contains e, but not e-pre, and Update is
--true, else Update is false.

procedure UPDATE_EDGE(G: in out DGRAPH;
                      FROMKEY, TOKEY: in KEY_TYPE;
                      W: in STANDARD_ELEMENT;
                      SUCCESS: out BOOLEAN);

procedure RETRIEVE_NODE(G: in out DGRAPH;
                        TKEY: in KEY_TYPE;
                        E: out STANDARD_ELEMENT;
                        SUCCESS: out BOOLEAN);

--pre - None.
--post - If G-pre contains element E-pre  with key value
--KEY1, then E is E-pre and RETRIEVE returns true, else
--RETRIEVE is false.

procedure CREATE(G: in out DGRAPH;
                 SUCCESS: out BOOLEAN);

--pre - None.
--post - If a graph can be created than G is an empty
--graph and CREATE is true, else CREATE is false.

procedure KILL(G: in out DGRAPH);

--pre - None.
--post - GRAPH G does not exist.

procedure SET_WAITING(G: DGRAPH);

--pre - graph must be fully connected
```

```ada
   procedure BREADTH_FIRST_SEARCH(G: DGRAPH);

   procedure DEPTH_FIRST_SEARCH(G: DGRAPH);


private

   type STATUS is (WAITING, READY, PROCESSED);
   type GRAPH_NODE;
   type NODE_PTR is access GRAPH_NODE;
   type EDGE_NODE;
   type EDGE_PTR is access  EDGE_NODE;

   type GRAPH_NODE is
     record
       ELT:  STANDARD_ELEMENT;
       NODE_KEY: KEY_TYPE;
       NEXT_NODE: NODE_PTR;
       EDGE_HEAD: EDGE_PTR;
       NODE_STATUS    : STATUS;
     end record;

   type EDGE_NODE is
     record
       DESTIN_KEY: KEY_TYPE;
       NEXT_EDGE: EDGE_PTR;
       WEIGHT: STANDARD_ELEMENT;
       EDGE_STATUS: STATUS;
     end record;

   type DGRAPH_TYPE is
     record
       HEAD, TAIL: NODE_PTR;
       SENTINEL  : NODE_PTR;
     end record;


   type DGRAPH  is access DGRAPH_TYPE;

end GRAPH2_ADT;


with UNCHECKED_DEALLOCATION;
with QUEUES2;

package body GRAPH2_ADT is

   package MY_QUEUE is new QUEUES2(NODE_PTR);
   use MY_QUEUE;
```

136

```
procedure FIND_NODE(G: in out DGRAPH; TKEY: in KEY_TYPE;
                    PTR: out NODE_PTR) is

   GP: NODE_PTR;

begin
  if G.HEAD /= null then
    G.SENTINEL.NODE_KEY := TKEY;
    G.TAIL.NEXT_NODE := G.SENTINEL;
    GP := G.HEAD;
    while TKEY /= GP.NODE_KEY loop
      GP := GP.NEXT_NODE;
      if GP = null then
        PTR := GP;
        return;
      end if;
    end loop;
    G.TAIL.NEXT_NODE := null;
    if GP /= G.SENTINEL then
      PTR := GP;
      return;
    end if;
  end if;
  PTR := null;
end FIND_NODE;



procedure INSERT_NODE(G: in out DGRAPH;
                      TKEY: in KEY_TYPE;
                      E: in STANDARD_ELEMENT;
                      SUCCESS: out BOOLEAN) is

   GP: NODE_PTR;
   AUX: NODE_PTR;
   OK: BOOLEAN;

   procedure CREATE_NODE(P: in out NODE_PTR;
                         TKEY: in KEY_TYPE;
                         E: in STANDARD_ELEMENT;
                         SUCCESS: out BOOLEAN) is

   begin
     P := new GRAPH_NODE;
     SUCCESS := TRUE;
     P.ELT := E;
     P.NODE_KEY := TKEY;
   exception
     when STORAGE_ERROR =>
       SUCCESS := FALSE;
   end CREATE_NODE;
```

```
      begin
        FIND_NODE(G, TKEY, AUX);
        if AUX = null then
          CREATE_NODE(GP, TKEY, E, OK);
          if OK then
            if G.HEAD = null then
              G.HEAD := GP;
            else
              G.TAIL.NEXT_NODE := GP;
            end if;
            G.TAIL := GP;
            SUCCESS := TRUE;
          end if;
        else
          SUCCESS := FALSE;
        end if;
      end INSERT_NODE;

      procedure FIND_EDGE(G: in out DGRAPH;
                          TKFROM, TKTO: in KEY_TYPE;
                          EP: out EDGE_PTR;
                          SUCCESS: out BOOLEAN) is

        AUX_EDGE_PTR: EDGE_PTR;
        AUX_NODE_PTR: NODE_PTR;

      begin
        SUCCESS := FALSE;
        FIND_NODE(G, TKFROM, AUX_NODE_PTR);
        if AUX_NODE_PTR =null then
          SUCCESS := FALSE;
        else
          if AUX_NODE_PTR.EDGE_HEAD /= null then
            AUX_EDGE_PTR := AUX_NODE_PTR.EDGE_HEAD;
            while AUX_EDGE_PTR /= null loop
              if AUX_EDGE_PTR.DESTIN_KEY = TKTO then
                SUCCESS := TRUE;
                EP := AUX_EDGE_PTR;
                exit;
              end if;
              AUX_EDGE_PTR := AUX_EDGE_PTR.NEXT_EDGE;
            end loop;
          else
            SUCCESS := FALSE;
          end if;
        end if;
      end FIND_EDGE;

      procedure INSERT_EDGE(G: in out DGRAPH;
                            FROM_KEY, TO_KEY: in KEY_TYPE;
                            W: in STANDARD_ELEMENT;
```

```
                        SUCCESS: out BOOLEAN) is

   ALREADY_EXISTS: BOOLEAN;
   AUX_NODE_PTR: NODE_PTR;
   AUX_EDGE_PTR1, AUX_EDGE_PTR2: EDGE_PTR;
   EP: EDGE_PTR;


begin
   FIND_EDGE(G, FROM_KEY, TO_KEY, EP, ALREADY_EXISTS);
   if ALREADY_EXISTS then
     SUCCESS := FALSE;
   else
     EP := new EDGE_NODE;
     EP.DESTIN_KEY := TO_KEY;
     EP.WEIGHT := W;
     EP.NEXT_EDGE := null;
     FIND_NODE(G, FROM_KEY, AUX_NODE_PTR);
     if AUX_NODE_PTR.EDGE_HEAD = null then
       AUX_NODE_PTR.EDGE_HEAD := EP;
     else
       AUX_EDGE_PTR1 := AUX_NODE_PTR.EDGE_HEAD;
       while AUX_EDGE_PTR1 /= null loop
         AUX_EDGE_PTR2 := AUX_EDGE_PTR1;
         AUX_EDGE_PTR1 := AUX_EDGE_PTR1.NEXT_EDGE;
       end loop;
       AUX_EDGE_PTR2.NEXT_EDGE := EP;
     end if;
   end if;
end INSERT_EDGE;

procedure RETRIEVE_EDGE(G: in out DGRAPH;
                        FROM_KEY, TO_KEY: in KEY_TYPE;
                        W: out STANDARD_ELEMENT;
                        SUCCESS: out BOOLEAN) is

   EP: EDGE_PTR;
   OK: BOOLEAN;


begin
   FIND_EDGE(G, FROM_KEY, TO_KEY, EP, OK);
   if OK then
     SUCCESS := TRUE;
     W := EP.WEIGHT;
   else
     SUCCESS := FALSE;
   end if;
end RETRIEVE_EDGE;
```

139

```ada
   procedure FREE_GRAPH_NODE is new
UNCHECKED_DEALLOCATION (GRAPH_NODE,NODE_PTR);

   procedure DELETE_NODE(G: in out DGRAPH;
                         TKEY: in KEY_TYPE;
                         SUCCESS: out BOOLEAN) is

     AUX_NODE_PTR: NODE_PTR;
     EP: EDGE_PTR;
     OK: BOOLEAN;
     GP: NODE_PTR;
     PREVIOUS: NODE_PTR;
     PTR: NODE_PTR;

   begin
     if G.HEAD = null then
       SUCCESS := FALSE;
     else
       --first delete all edges towards this node
       AUX_NODE_PTR := G.HEAD;
       if AUX_NODE_PTR.NODE_KEY /= TKEY then
         DELETE_EDGE(G, AUX_NODE_PTR.NODE_KEY, TKEY, OK);
       end if;
       while AUX_NODE_PTR.NEXT_NODE /= null loop
         AUX_NODE_PTR := AUX_NODE_PTR.NEXT_NODE;
         if AUX_NODE_PTR.NODE_KEY /= TKEY then
           DELETE_EDGE(G, AUX_NODE_PTR.NODE_KEY, TKEY, OK);
         end if;
       end loop;

       --delete the node
       G.SENTINEL.NODE_KEY := TKEY;
       G.TAIL.NEXT_NODE := G.SENTINEL;
       GP := G.HEAD;
       PREVIOUS := GP;
       while TKEY /= GP.NODE_KEY loop
         PREVIOUS := GP;
         GP := GP.NEXT_NODE;
       end loop;
       G.TAIL.NEXT_NODE := null;
       if GP /= G.SENTINEL then
         PTR := GP;
         if PTR = PREVIOUS then
           G.HEAD := PTR.NEXT_NODE;
           FREE_GRAPH_NODE(PTR);
         else
           PREVIOUS.NEXT_NODE := PTR.NEXT_NODE;
           PTR.NEXT_NODE := null;
           FREE_GRAPH_NODE(PTR);
         end if;
```

140

```
              SUCCESS := TRUE;
          else
              SUCCESS := FALSE;
          end if;
        end if;
   end DELETE_NODE;

   procedure FREE_EDGE_NODE is new
UNCHECKED_DEALLOCATION (EDGE_NODE,EDGE_PTR);

   procedure DELETE_EDGE(G: in out DGRAPH;
                          TKFROM, TKTO: in KEY_TYPE;
                          SUCCESS: out BOOLEAN) is

     AUX_NODE_PTR: NODE_PTR;
     AUX_EDGE_PTR: EDGE_PTR;
     EP: EDGE_PTR;
     PREVIOUS: EDGE_PTR;
     OK: BOOLEAN;

   begin
     OK := FALSE;
     FIND_NODE(G, TKFROM, AUX_NODE_PTR);
     if AUX_NODE_PTR = null then
       OK := FALSE;
     else
       AUX_EDGE_PTR := AUX_NODE_PTR.EDGE_HEAD;
       PREVIOUS := AUX_EDGE_PTR;
       while AUX_EDGE_PTR /= null loop
         if AUX_EDGE_PTR.DESTIN_KEY = TKTO then
           OK := TRUE;
           EP := AUX_EDGE_PTR;
           exit;
         end if;
         PREVIOUS := AUX_EDGE_PTR;
         AUX_EDGE_PTR := AUX_EDGE_PTR.NEXT_EDGE;
       end loop;
       if OK then
         if PREVIOUS = AUX_EDGE_PTR then
           AUX_NODE_PTR.EDGE_HEAD := AUX_EDGE_PTR.NEXT_EDGE;
           FREE_EDGE_NODE(AUX_EDGE_PTR);
         else
           PREVIOUS.NEXT_EDGE := AUX_EDGE_PTR.NEXT_EDGE;
           AUX_EDGE_PTR.NEXT_EDGE := null;
           FREE_EDGE_NODE(AUX_EDGE_PTR);
         end if;
       end if;
     end if;
     SUCCESS := OK;
   end DELETE_EDGE;
```

141

```
procedure UPDATE_NODE(G: in out DGRAPH;
                      TKEY: in KEY_TYPE;
                      E: in STANDARD_ELEMENT;
                      SUCCESS: out BOOLEAN) is

   PTR: NODE_PTR;

begin
   FIND_NODE(G, TKEY, PTR);
   if PTR = null then
      SUCCESS := FALSE;
   else
      PTR.ELT := E;
      SUCCESS := TRUE;
   end if;
end UPDATE_NODE;


procedure UPDATE_EDGE(G: in out DGRAPH;
                      FROMKEY, TOKEY: in KEY_TYPE;
                      W: in STANDARD_ELEMENT;
                      SUCCESS: out BOOLEAN) is

   EP: EDGE_PTR;
   OK: BOOLEAN;

begin
   FIND_EDGE(G, FROMKEY, TOKEY, EP, OK);
   if OK then
      EP.WEIGHT := W;
      SUCCESS := TRUE;
   else
      SUCCESS := FALSE;
   end if;
end UPDATE_EDGE;

procedure RETRIEVE_NODE(G: in out DGRAPH;
                        TKEY: in KEY_TYPE;
                        E: out STANDARD_ELEMENT;
                        SUCCESS: out BOOLEAN) is

   PTR: NODE_PTR;

begin
   FIND_NODE(G, TKEY, PTR);
   if PTR = null then
      SUCCESS := FALSE;
   else
      E := PTR.ELT;
      SUCCESS := TRUE;
   end if;
```

```
end RETRIEVE_NODE;

procedure CREATE(G: in out DGRAPH;
                 SUCCESS: out BOOLEAN) is

begin
  G := new DGRAPH_TYPE;
  G.SENTINEL := new GRAPH_NODE;
  SUCCESS := TRUE;
exception
  when STORAGE_ERROR =>
    SUCCESS := FALSE;
end CREATE;

procedure KILL(G: in out DGRAPH) is

  BL: BOOLEAN;

begin
  while G.HEAD /= null loop
    DELETE_NODE(G, G.HEAD.NODE_KEY, BL);
  end loop;
end KILL;

procedure SET_WAITING(G: DGRAPH) is

  AUX_NODE_PTR: NODE_PTR := G.HEAD;
  AUX_EDGE_PTR: EDGE_PTR;

begin
  while AUX_NODE_PTR /= null loop
    AUX_NODE_PTR.NODE_STATUS := WAITING;
    AUX_EDGE_PTR := AUX_NODE_PTR.EDGE_HEAD;
    while AUX_EDGE_PTR /= null loop
      AUX_EDGE_PTR.EDGE_STATUS := WAITING;
      AUX_EDGE_PTR := AUX_EDGE_PTR.NEXT_EDGE;
    end loop;
    AUX_NODE_PTR := AUX_NODE_PTR.NEXT_NODE;
  end loop;
end SET_WAITING;

procedure BREADTH_FIRST_SEARCH(G: DGRAPH) is

  Q: QUEUE;
  SUCCESS: BOOLEAN;
  GP: NODE_PTR;
  AUX_GRAPH: DGRAPH := G;

  procedure VISIT(P: in out NODE_PTR) is -- Visit a graph
  --node
```

```
      EP: EDGE_PTR;
      AUX_NODE_PTR: NODE_PTR;

    begin
      ENQUEUE(Q, P);
      while not EMPTY(Q) loop
        SERVE(Q, P);
        PROCESS(P.ELT, P.NODE_KEY);
        P.NODE_STATUS := PROCESSED;
        EP := P.EDGE_HEAD;
        while EP /= null loop -- Consider all neighbors
          FIND_NODE(AUX_GRAPH, EP.DESTIN_KEY, AUX_NODE_PTR);
          if AUX_NODE_PTR.NODE_STATUS = WAITING then
            ENQUEUE(Q, AUX_NODE_PTR);
            AUX_NODE_PTR.NODE_STATUS := READY;
          end if;
          EP := EP.NEXT_EDGE;
        end loop;
      end loop;
    end VISIT;

  begin
    CREATE(Q, SUCCESS);
    if not SUCCESS then
      raise QUEUE_ERROR;
    end if;
    SET_WAITING(G);
    GP := G.HEAD;
    while GP /= null loop
      if GP.NODE_STATUS = WAITING then
        VISIT(GP);
      else
        GP := GP.NEXT_NODE;
      end if;
    end loop;
  end BREADTH_FIRST_SEARCH;


  procedure DEPTH_FIRST_SEARCH(G: DGRAPH) is

    GP: NODE_PTR;

    procedure VISIT(MY_P: NODE_PTR) is

      P: NODE_PTR := MY_P;
      EP: EDGE_PTR;
      AUX_NODE_PTR: NODE_PTR;
      AUX_GRAPH: DGRAPH := G;

    begin
      PROCESS(P.ELT, P.NODE_KEY);
```

```
        P.NODE_STATUS := PROCESSED;
        EP := P.EDGE_HEAD;
        while EP /= null loop
          FIND_NODE(AUX_GRAPH, EP.DESTIN_KEY, AUX_NODE_PTR);

          if AUX_NODE_PTR.NODE_STATUS = WAITING then
            VISIT(AUX_NODE_PTR);
          end if;
          EP := EP.NEXT_EDGE;
        end loop;
      end VISIT;

    begin
      SET_WAITING(G);
      GP := G.HEAD;
      while GP /= null loop
        if GP.NODE_STATUS = WAITING then
          VISIT(GP);
        else
          GP := GP.NEXT_NODE;
        end if;
      end loop;
    end DEPTH_FIRST_SEARCH;

end GRAPH2_ADT;
```

C. QUEUES2.ADA

```
generic
  type STANDARD_ELEMENT is private;

package QUEUES2 is

  --Elements: Although the elements can be a variety of
  --types, for concreteness we assume that they are of type
  --standard element.

  --Structure: The structure is a mechanism for relating the
  --elements that allows determination of their order of
  --arrival into the queue.

  --Domain    : The number of elements in the queue is
  --bounded.

  type QUEUE is private;
  QUEUE_ERROR: exception;

  --Operations: There are six operations. Occasionally in
  --the postcondition we must reference the value of the
```

```
      --queue immediately before execution of the operation. We
      --use Q-pre as notation for this operation.

      procedure ENQUEUE(Q: in out QUEUE; E: in

                        STANDARD_ELEMENT);

      --pre:  The size of Q is less than its bound.
      --post: Q includes E as its most recently arrived element.

      procedure SERVE(Q: in out QUEUE; E: out STANDARD_ELEMENT);

      --pre: Q is not empty.
      --post: E is at least recently arrived element of Q-pre;
      --Q does not contain E.

      function LENGTH(Q: in QUEUE) return NATURAL;

      --post: Length is the number of elements in Q.

      function FULL(Q: in QUEUE) return BOOLEAN;

      --post: If the size of Q is less than its bound then Full
      --is false,
      --else full is true.

      function EMPTY(Q: in QUEUE) return BOOLEAN;

      --post: If the size of Q is zero returns TRUE else returns
      --FALSE.

      procedure CREATE(Q: in out QUEUE; SUCCESS: out BOOLEAN);
      --post: If a queue can be created, then q exists and is
      --empty and create is true, else create is false.

      procedure KILL(Q: in out QUEUE);

      --post: Q-pre does not exist.
private
   type NODE;
   type NODE_POINTER is access NODE;

   type NODE is
     record
       ELEMENT: STANDARD_ELEMENT;
       NEXT    : NODE_POINTER;
     end record;

   type QUEUE_INSTANCE is
     record
```

146

```ada
      HEAD,
      TAIL: NODE_POINTER;
      SIZE: NATURAL := 0;
    end record;

  type QUEUE is access QUEUE_INSTANCE;

end QUEUES2;


with UNCHECKED_DEALLOCATION;

package body QUEUES2 is

  procedure ENQUEUE(Q: in out QUEUE; E: in STANDARD_ELEMENT)
                                          is
    --Add element E to queue Q.

  begin
    if Q.SIZE = 0 then --insert new node into empty queue
      Q.TAIL := new NODE'(E, NULL);
      Q.HEAD := Q.TAIL;
    else
      Q.TAIL.NEXT := new NODE'(E, NULL); --link new node to
                                         --tail
      Q.TAIL := Q.TAIL.NEXT;
    end if;

    Q.SIZE := Q.SIZE + 1;
  exception
    when STORAGE_ERROR =>
      raise QUEUE_ERROR;
  end ENQUEUE;


  procedure DISPOSE is new UNCHECKED_DEALLOCATION(NODE,
  NODE_POINTER);

  procedure SERVE(Q: in out QUEUE; E: out STANDARD_ELEMENT)
                                        is
    --Retrieve and remove most recently added element.
    P: NODE_POINTER; --temporary pointer
  begin
    if Q.SIZE = 0 then --queue is empty
      raise QUEUE_ERROR;
    end if;

    E := Q.HEAD.ELEMENT; --extract top element
    P := Q.HEAD;
    Q.HEAD := Q.HEAD.NEXT;
```

```
      DISPOSE(P);
   Q.SIZE := Q.SIZE - 1;
end SERVE;


function LENGTH(Q: in QUEUE) return NATURAL is

   --return size of the queue
begin
   return Q.SIZE;
end LENGTH;


function FULL(Q: in QUEUE) return BOOLEAN is


   --Is Q full?
   P: NODE_POINTER; --temporary pointer
begin
   P := new NODE;
   DISPOSE(P);
   return FALSE;
exception
   when STORAGE_ERROR =>
      return TRUE;
end FULL;


function EMPTY(Q: in QUEUE) return BOOLEAN is

--Is Q empty
begin
   return Q.SIZE = 0;
end EMPTY;


procedure CREATE(Q: in out QUEUE; SUCCESS: out BOOLEAN) is

--If a queue can be created, then do so and return
--SUCCESS = true else return SUCCESS = FALSE.
begin
   if Q = NULL then
     Q := new QUEUE_INSTANCE;
     SUCCESS := TRUE;
   else --storage has already been allocated for queue
     KILL(Q);
   end if;
exception
   when STORAGE_ERROR => -- out of memory
     SUCCESS := FALSE;
end CREATE;
```

148

```
     procedure KILL(Q: in out QUEUE) is

       P: NODE_POINTER;
     begin
       while Q.HEAD /= NULL loop
         P := Q.HEAD;
         Q.HEAD := Q.HEAD.NEXT;
         DISPOSE(P);
       end loop;
       Q.TAIL := NULL;
       Q.SIZE := 0;
     end KILL;

end QUEUES2;
```

D. SORT.ADA

```
generic
  type STANDARD_ELEMENT is private;
  type INDEX is range <>;
  with function "<" (E1,E2: STANDARD_ELEMENT) return
BOOLEAN;

package SORT_ADT is

  --Each of the procedures in this module, except heapsort,
  --sorts an array of standard elements into ascending
--order.Heapsort sorts into descending order.

  type SORT_ARRAY is array (INDEX range <>) of
STANDARD_ELEMENT;

  procedure SELECT_SORT (R: in out SORT_ARRAY);
  procedure EXCHANGE_SORT(R: in out SORT_ARRAY);
  procedure INSERT_SORT (R: in out SORT_ARRAY);
  procedure QS_2 (R: in out SORT_ARRAY);
  procedure QS_3 (R: in out SORT_ARRAY);
  procedure MERGE_SORT (R: in out SORT_ARRAY);

end SORT_ADT;


package body SORT_ADT is

  procedure SWAP (EL1, EL2: in out STANDARD_ELEMENT) is

    TEMP_EL2: STANDARD_ELEMENT := EL2;
```

```
begin
  EL2 := EL1;
  EL1 := TEMP_EL2;
end SWAP;

function ">" (E1, E2: STANDARD_ELEMENT) return BOOLEAN is

begin
  return (E1 /= E2) and not (E1 < E2);
end ">";

function ">=" (E1,E2: STANDARD_ELEMENT) return BOOLEAN is

begin
  return (E1 > E2) or (E1 = E2);
end ">=";

function "<=" (E1, E2: STANDARD_ELEMENT) return BOOLEAN is

begin
  return (E1 < E2) or (E1 = E2);
end "<=";

procedure SELECT_SORT (R: in out SORT_ARRAY) is

  SMALL: INDEX;

begin
  for K in R'FIRST..INDEX'PRED(R'LAST) loop
    SMALL := K;
    for J in INDEX'SUCC(K)..R'LAST loop
      if R(J) < R(SMALL) then
        SMALL := J;
      end if;
    end loop;
    SWAP (R(K), R(SMALL));
  end loop;
end SELECT_SORT;

procedure EXCHANGE_SORT (R: in out SORT_ARRAY) is

  SORTED: BOOLEAN;
  K:      INDEX;

begin
  K := R'LAST;
  SORTED := FALSE;

  while (K > R'FIRST) and (not SORTED) loop
    SORTED := TRUE;
```

```
        for J in R'FIRST..INDEX'PRED(K) loop
          if R(J) > R(INDEX'SUCC(J)) then
            SORTED := FALSE;
          end if;
        end loop;
        K := INDEX'PRED(K);
    end loop;
end EXCHANGE_SORT;

procedure INSERT_SORT (R: in out SORT_ARRAY) is

  SAVE: STANDARD_ELEMENT;
  J: INDEX;

begin
  if R'LENGTH > 1 then
    for K in reverse R'FIRST..INDEX'PRED(R'LAST) loop
      J := INDEX'SUCC(K);
      SAVE := R(K);
      while (SAVE > R(J)) loop
        R(INDEX'PRED(J)) := R(J);
        if J < R'LAST then
          J := INDEX'SUCC(J);
        else
          exit;
        end if;
      end loop;
      if SAVE > R(J) then
        R(J) := SAVE;
      else
        R(INDEX'PRED(J)) := SAVE;
      end if;
    end loop;
  end if;
end INSERT_SORT;

procedure QS_2 (R: in out SORT_ARRAY) is

  procedure QUICK_2(LEFT, RIGHT: INDEX) is

    --post - Element R(LEFT)-pre is in the sorted
  --position, say k, with R(LEFT) ... R(K-1) less then
--R(K) and R(K+1)... R(RIGHT) larger than R(K).

    J,K: INDEX;

  begin
    if LEFT < RIGHT then
      J := LEFT;
      K := RIGHT;
      if R(LEFT) > R(RIGHT) then
```

```
                  SWAP(R(LEFT), R(RIGHT));
             end if;
             loop --until J > K
               loop --until R(J) >= R(LEFT)
                 J := INDEX'SUCC(J);
                 exit when R(J) >= R(LEFT);
               end loop;
               loop --until R(K) <= R(LEFT)
                 K := INDEX'PRED(K);
                 exit when R(K) <= R(LEFT);
               end loop;
               if J < K then
                 SWAP( R(J), R(K));
               end if;
               exit when J > K;
             end loop;
             SWAP(R(LEFT), R(K));
             if K > INDEX'FIRST then
               QUICK_2 (LEFT, INDEX'PRED(K));
             end if;
             QUICK_2 (INDEX'SUCC(K), RIGHT);
           end if;
         end QUICK_2;

  begin  --QS_2
    QUICK_2(R'FIRST, R'LAST);
  end QS_2;

  procedure QS_3 (R: in out SORT_ARRAY) is --Quicksort

     procedure QUICK3(LEFT, RIGHT: INDEX) is

        J,K : INDEX;
     begin
       if LEFT < RIGHT then
         --Median of 3 modification
         SWAP(R(INDEX'VAL((R'LENGTH+1)/2)),
         R(INDEX'SUCC(LEFT)));
         if R(INDEX'SUCC(LEFT)) > R(RIGHT) then
           SWAP(R(INDEX'SUCC(LEFT)), R(RIGHT));
         end if;
         if R(LEFT) > R(RIGHT) then
           SWAP(R(LEFT), R(RIGHT));
         end if;
         if R(INDEX'SUCC(LEFT)) > R(LEFT) then
           SWAP(R(INDEX'SUCC(LEFT)), R(LEFT));
         end if;

         --After Median of 3 mod. is complete, the smallest
         --sample value will be in position LEFT + 1; the
```

152

```
               --largest sample value will be in position RIGHT;
               --and the median sample value will be in
               --position LEFT.

               J := INDEX'SUCC(LEFT);
               K := RIGHT;

               loop
                 loop --Advance J right until a value greater than
                   --R(LEFT) is found
                   J := INDEX'SUCC(J);
                   exit when R(J) >= R(LEFT);
                 end loop;
                 loop --Advance K left until a value less than
                   --R(LEFT) is found.
                   K := INDEX'PRED(K);
                   exit when R(K) <= R(LEFT);
                 end loop;
                 if J < K then -- Swap values
                   SWAP(R(J), R(K));
                 end if;
                 exit when J > K;
               end loop;
               SWAP (R(LEFT), R(K)); --Put median value into its
                                     --sorted position

               --QUICK SORT remaining sublist partitions if they
               --contain 10 or more elements.

               if (INDEX'POS(K) - INDEX'POS(LEFT)) > 10 then
                 QUICK3(LEFT, INDEX'PRED(K));
               end if;
               if (INDEX'POS(RIGHT) - INDEX'POS(K)) > 10 then
                 QUICK3(INDEX'SUCC(K), RIGHT);
               end if;
           end if;
         end QUICK3;

begin --QS 3
  QUICK3 (R'FIRST, R'LAST);
  INSERT_SORT(R);
end QS_3;

procedure MERGE(L: NATURAL; R,T: in out SORT_ARRAY) is

   --pre - Array R contains sorted sublists of length L.
   --post- Array T contains sorted sublists of lenght 2L.

   Q, K1, K2, END1, END2: NATURAL;

begin
```

```
      if L >= R'LENGTH then
        T := R;
        return;
      end if;
      K1 := 1; -- K1 and K2 are indexes for the sublists to
      K2 := L + 1; -- be merged
      Q := 0;
      loop -- Repeat
        END1 := K1 + L - 1;
        if END1 > R'LENGTH then --Mark the ends of the
                                --sublists to be
          END1 := R'LENGTH;       --merged.
        else
          END2 := K2 + L - 1;
          if END2 > R'LENGTH then
            END2 := R'LENGTH;
          end if;
          while (K1 <= END1) and (K2 <= END2) loop
            if R(INDEX'VAL(K1)) <= R(INDEX'VAL(K2)) then
              T(INDEX'VAL(Q+1)) := R(INDEX'VAL(K1));
              K1 := K1 + 1;
            else
              T(INDEX'VAL(Q+1)) := R(INDEX'VAL(K2));
              K2 := K2 + 1;
            end if;
            Q := Q + 1;
          end loop;
        end if;
        if K1 <= END1 then          --Tack on elements from the
                                    --sublist with
          for K in K1..END1 loop --more elements
            T(INDEX'VAL(Q+1)) := R(INDEX'VAL(K));
            Q := Q + 1;
          end loop;
          K1 := END1 + 1;
        else
          for K in K2..END2 loop
            T(INDEX'VAL(Q+1)) := R(INDEX'VAL(K));
            Q := Q+1;
            K2 := END2 + 1;
          end loop;
          K2 := END2 + 1;
        end if;
        K1 := K2;                           --Set indexes for the next
                                            --pair of sublists
        K2 := K1 + L;
        exit when K1 > R'LENGTH;
      end loop;

end MERGE;
```

```
procedure MERGE_SORT (R: in out SORT_ARRAY) is

   T: SORT_ARRAY(R'FIRST..R'LAST);
   L: NATURAL;

begin
   L := 1;
   if R'LENGTH >= 2 then
      loop
         MERGE(L, R, T);
         L := 2*L;
         MERGE(L, T, R);
         L := 2*L;
         exit when L >= (R'LENGTH+1)/2;
      end loop;
   end if;
   if L < R'LENGTH then   -- If necessary, a final merge.
      MERGE(L, R, T);
      R := T;
   end if;
end MERGE_SORT;

end SORT_ADT;
```

APPENDIX D: TASK ALLOCATION ADA PROCEDURES

A. STATICAL.ADA

```
with RANDOM;
with TEXT_IO;
use TEXT_IO;
with QUEUES2; -- linked list implementation of queues
with DISCRETE_SET; -- sets are used as the main data
-- structures to control the allocation
with SORT_ADT; -- several schemes of sorting
with GRAPH2_ADT; -- adjacency list implementation of
--directed acyclic graphs


procedure STATICAL is


-- instantation of I/O routines
package INTEGER_INOUT is new INTEGER_IO(INTEGER);
use INTEGER_INOUT;
package FLOAT_INOUT is new FLOAT_IO(FLOAT);
use FLOAT_INOUT;


-- defines the maximum number of tasks to be used with this
-- program
MAX_PROCESS: constant INTEGER := 25;
NUM_OF_TASKS: INTEGER; -- global variable that holds the
-- number of tasks in a task flow graph

MAX_INDEX: INTEGER; -- global variable that holds the
-- maximum valid index for the heuristic array
-- defines the maximum number of components that can be used
-- by the array that holds the values calculated by
-- heuristic for each possible graph edge
MAX_ARRAY: constant INTEGER :=
(MAX_PROCESS*(MAX_PROCESS-1))/2;


-- MY_ATOM is basically the task identifier
type MY_ATOM is range 1..MAX_PROCESS;
```

```
package MY_DIS is new DISCRETE_SET(MY_ATOM);
use MY_DIS;

-- data structures used for the calculation of ancestor and
-- parent arrays
type PRED_ARRAY is array (2..MY_ATOM'LAST) of SET;
PARENT_ARRAY: PRED_ARRAY := (others => MY_DIS.CREATE);
ANCESTOR_ARRAY: PRED_ARRAY := (others => MY_DIS.CREATE);

-- record that defines the structure of the information
-- stored on each task node and on each task edge
type MY_ELEMENT is
  record
    INFO: FLOAT; -- may represent execution costs(node) or
                 -- communication costs(edge)
    ACCUMULATOR: FLOAT := 0.0; -- accumulator used for the
    -- calculation of the longest path cost and also by
    -- heuristics that take into account the displacement in
    -- time from the root task
    PROCESS_SET: SET; -- stores information related with the
    -- longest path ,that is the set of task identifiers
  -- that compose the longest path end record;

    -- record that defines the structure of the information
    -- stored in the
    -- array that holds the results of heuristic functions
    -- applications
  type MY_RECORD is
    record
      PAIR_SET: SET := MY_DIS.CREATE; -- edge defined as a
      -- pairset of task identifiers
      INDEX_FROM: INTEGER;   -- source node
      INDEX_TO:   INTEGER;   -- destination node
      EFROM    : FLOAT;      -- execution time of the source
      -- node
      ETO      : FLOAT;      -- execution time of the
      -- destination node
      COMM     : FLOAT;      -- communications cost
      HINFO    : FLOAT;      -- result of the heuristic
      -- function
    end record;

  -- defines the number of processors used in the allocation
  -- scheme
  type PROCESSOR is (P1, P2, P3, P4, P5, P6, P7, P8, P9,
  P10);

  -- gets the number of processors to be used
  LAST_PROCESSOR: PROCESSOR;

  -- Array type that has one set per processor. Each set
```

```ada
      -- defines which tasks are allocated to that processor
      type PROCESSOR_ARRAY is array(PROCESSOR) of SET;

      -- processor array for application of heuristic
      P_ARRAY: PROCESSOR_ARRAY := (others => MY_DIS.CREATE);


      -- Keeps track of the accumulated absolute utilization of
      -- each processor
      -- that composes the network
      type UTILIZATION_ARRAY is array(PROCESSOR) of FLOAT;

      -- utilization array for applycation of heuristic1
      U_ARRAY: UTILIZATION_ARRAY := (others => 0.0);

      -- controls what are the tasks already allocated in
      -- accordance with heuristic1
      ALLOCATED_TASKS: SET := MY_DIS.CREATE;

      UNI_EXE_COST: FLOAT := 0.0; -- uniprocessor execution time
      -- of the data flow
      AUX_COST: FLOAT := 0.0;   -- aux. variable to compute
      -- uniprocessor execution time
      AUX_COUNT: INTEGER := 0; -- global variable used by the
      -- procedure printout to count the number of tasks in a
      -- graph
      POTENTIAL_SPEED_UP: FLOAT; -- speed-up that could be
      -- achieved if the minimum cost were equal to the sum of
-- the tasks that compose the longest path

      -- Record used by the procedure SCHEDULE
      type EXECUTION_INFO is
        record

          START_TIME: FLOAT;    -- task or communication start
          --time
          END_TIME:   FLOAT;    -- task or communication end time
          TASK_ID:    MY_ATOM; -- task identifier
          IS_COMM:    BOOLEAN; -- communication or task
          -- execution time
          TASK_FROM:  MY_ATOM; -- message sender
          TASK_DESTIN: MY_ATOM; -- message receiver
        end record;


      -- Queue that is used by the longest path cost algorithm
      -- included in the procedure CONSTRUCT_TASK_FLOW
      package NEW_QUEUES is new QUEUES2(MY_ELEMENT);


      --Array that keeps the schedule for every processor
```

```
type SCHEDULE_ARRAY is array(MY_ATOM) of EXECUTION_INFO;
THE_SCHEDULE: SCHEDULE_ARRAY;

--Array that is used by the scheduler
type TIME_ARRAY is array(PROCESSOR) of FLOAT;
MY_TIME: TIME_ARRAY;



-- Used for instantiation of the package GRAPH2_ADT
procedure PRINT_OUT(ELEMENT: in MY_ELEMENT; ELEMENT_KEY:
in INTEGER);

--Instantiation of the package GRAPH2_ADT
package MY_DGRAPHS is new
GRAPH2_ADT(MY_ELEMENT,INTEGER,PRINT_OUT);
use MY_DGRAPHS;

-- Used for instantiation of package SORT_ADT
function LESS_THAN (E1, E2: MY_RECORD) return BOOLEAN;

-- Instantiation of the package SORT_ADT
package NEW_SORT is new SORT_ADT(MY_RECORD, INTEGER,
LESS_THAN);
use NEW_SORT;

DG: DGRAPH;                              -- directed graph
-- abstract data type
RESULT_SET: SET := MY_DIS.CREATE;   -- set tha contains the
-- tasks that compose the longest path cost
HEUR_ARRAY: SORT_ARRAY(1..MAX_ARRAY);-- array with values
-- that were calculated by heuristic function

-- Procedure used in the instantiation of the package
-- GRAPH2_ADT.
-- This procedure prints the element key(task identifier)
-- and the execution cost associated with the task.It also
-- accumulates the uniprocessor execution time and counts
-- the number  of tasks in the task flow. This procedure
-- is used by the traversals depth_first_search
-- and breadth_first_search of the package MY_DGRAPHS.
procedure PRINT_OUT(ELEMENT: in MY_ELEMENT; ELEMENT_KEY:
in INTEGER) is


begin
  PUT(ELEMENT_KEY, WIDTH => 4); --prints task identifier
  PUT("  ");
  PUT(ELEMENT.INFO, FORE => 2, AFT => 2, EXP => 0);
  --prints task execution cost
  NEW_LINE;
```

```
      if AUX_COUNT < MAX_PROCESS then
        AUX_COUNT := AUX_COUNT + 1; -- counts the number of
        -- tasks
        AUX_COST := AUX_COST + ELEMENT.INFO;--accumulates
        -- uniprocessor execution cost
      end if;
end PRINT_OUT;


-- Function used for instantiation of the package SORT_ADT

function LESS_THAN (E1, E2: MY_RECORD) return BOOLEAN is


begin
   if E1.HINFO < E2.HINFO then
     return TRUE;
   else
     return FALSE;
   end if;
end LESS_THAN;

-- Prints all atoms that compose a set
procedure PRINT_SET(A: SET) is

begin
   for I in 1..MAX_PROCESS loop
     if MY_DIS.MEMBER(MY_ATOM(I), A) then
       PUT(I, WIDTH => 3);
     end if;
   end loop;
   NEW_LINE;
end PRINT_SET;

-- Prints one processor
procedure PRINT_PROCESSOR(PROC: PROCESSOR; L_PROC:
PROCESSOR) is

AUX_INDEX: INTEGER := 0;

begin
   for I in PROCESSOR'FIRST..L_PROC loop
     AUX_INDEX := AUX_INDEX + 1;
     if I = PROC then
       PUT("Tasks Allocated to P");
       PUT(AUX_INDEX, WIDTH => 1);
       PUT_LINE("/Utilisation");
       exit;
     end if;
   end loop;
```

```
    end PRINT_PROCESSOR;


    -- Finds in which processor the task is allocatted
    procedure FIND_PROCESSOR(P: in PROCESSOR_ARRAY;
                             IN_ATOM: in MY_ATOM;
                             L_PROC: in PROCESSOR;
                             SUCCESS: out BOOLEAN;
                             OUT_PROCESSOR: out PROCESSOR) is

    OK: BOOLEAN := FALSE;

    begin
       for I in PROCESSOR'FIRST..L_PROC loop
         OK := MY_DIS.MEMBER(IN_ATOM, P(I));
         if OK then
           SUCCESS := OK;
           OUT_PROCESSOR := I;
           exit;
         end if;
       end loop;
    end FIND_PROCESSOR;



    -- Constructs one example of task flow and returns a set
    -- that contains all tasks that are in its longest path
    -- cost.
    procedure CONSTRUCT_TASK_FLOW(G: in out DGRAPH;
                             OUT_SET: in out SET)
                                   is separate;

    -- Calculate heuristics for each edge of the directed
    -- graph giving the results in TEMP_ARRAY in accordance
    -- with the heuristic function specified by
    -- HEUR.TEMP_ARRAY is sorted in increasing order using the
    -- values of the calculated heuristics
    procedure CALC_HEURISTIC(G: in out DGRAPH;
                        TEMP_ARRAY: in out SORT_ARRAY)
    is separate;


    -- Allocate tasks to processors.
    procedure ALLOCATE(G: in out DGRAPH; LAST_PROC: out
                       PROCESSOR;
                       TEMP_ARRAY: in out SORT_ARRAY;
                       TEMP_SET: in out SET; --longest path
                       -- cost set
                       PROC_ARRAY: in out PROCESSOR_ARRAY;
                       UTIL_ARRAY: in out UTILIZATION_ARRAY;
                       ALLOCATED_TASKS: in out SET) is separate;
```

161

```
   -- Schedule tasks on each processor
   procedure SCHEDULE(G: in out DGRAPH;
                      L_PROC: in PROCESSOR;
                      P: in out PROCESSOR_ARRAY;
                      S: in out SCHEDULE_ARRAY;
                      T: out TIME_ARRAY) is separate;

   -- Improves the task allocation
   procedure IMPROVE is separate;

begin
   CONSTRUCT_TASK_FLOW(DG,RESULT_SET);
   CALC_HEURISTIC(DG,HEUR_ARRAY);
   ALLOCATE(DG,LAST_PROCESSOR,HEUR_ARRAY,RESULT_SET,
   P_ARRAY,U_ARRAY,ALLOCATED_TASKS);
   SCHEDULE(DG,LAST_PROCESSOR,P_ARRAY,THE_SCHEDULE,MY_TIME);
   if LAST_PROCESSOR = P4 then
      IMPROVE;
   end if;

end STATICAL;
```

B. CTFLOW.ADA

```
separate(STATICAL)

procedure CONSTRUCT_TASK_FLOW(G: in out DGRAPH;
         OUT_SET: in out SET) is

 TITLE: STRING(1..80);
 TITLE_LENGTH:NATURAL;
 SUCCESS: BOOLEAN;
 FILE_NAME: STRING(1..30);
 NAME_LENGTH: NATURAL;
 INF: FILE_TYPE;
 MY_STRING: STRING(1..4);
 MY_NODE: INTEGER;
 MY_EXE: FLOAT;
 MY_FROM:INTEGER;
 MY_TO: INTEGER;
 MY_COST: FLOAT;

 COMIJ: MY_ELEMENT;
```

162

```
function CONVERT(E: FLOAT) return MY_ELEMENT is

MY_DATA: MY_ELEMENT;

begin
 MY_DATA.PROCESS_SET := MY_DIS.CREATE;
 MY_DATA.INFO := E;
 return MY_DATA;
end CONVERT;


procedure FIND_LONG_PATH(DG: in out DGRAPH; OUT_SET: in out
SET) is

E1, E2, E3: MY_ELEMENT;
SUCCESS: BOOLEAN;
AUX_SET: SET;
ACC: FLOAT;
MAX: FLOAT;
AUX_QUEUE: NEW_QUEUES.QUEUE;


begin
  NEW_QUEUES.CREATE(AUX_QUEUE, SUCCESS);
  for I in 1..MAX_PROCESS loop
    RETRIEVE_NODE(DG, I, E1, SUCCESS);
    if not MY_DIS.MEMBER(MY_ATOM(I), E1.PROCESS_SET) then
      E1.ACCUMULATOR := E1.ACCUMULATOR + E1.INFO;
      MY_DIS.INSERT(MY_ATOM(I), E1.PROCESS_SET);
      UPDATE_NODE(DG, I, E1, SUCCESS);
    end if;
    for J in 1..MAX_PROCESS loop
      if I /= J then
        RETRIEVE_EDGE(DG, I, J, E2, SUCCESS);
        if SUCCESS then
          RETRIEVE_NODE(DG, J, E2, SUCCESS);
          if SUCCESS and J=1 then
            RETRIEVE_NODE(DG, I, E3, SUCCESS);
            NEW_QUEUES.ENQUEUE(AUX_QUEUE, E3);
          else
            ACC := E1.ACCUMULATOR + E2.INFO;
            if ACC > E2.ACCUMULATOR then
              E2.ACCUMULATOR := ACC;
              E2.PROCESS_SET := E1.PROCESS_SET;
              MY_DIS.INSERT(MY_ATOM(J), E2.PROCESS_SET);
              UPDATE_NODE(DG, J, E2, SUCCESS);
            end if;
          end if;
        end if;
      end if;
```

```
        end loop;
      end loop;
      MAX := 0.0;
      while not NEW_QUEUES.EMPTY(AUX_QUEUE) loop
        NEW_QUEUES.SERVE(AUX_QUEUE, E1);
        if E1.ACCUMULATOR > MAX then
          MAX := E1.ACCUMULATOR;
          OUT_SET := E1.PROCESS_SET;
        end if;
      end loop;

  end FIND_LONG_PATH;


begin
  --creates the directed graph
  CREATE(G, SUCCESS);
  if not SUCCESS then
    raise DGRAPH_ERROR;
  end if;

  PUT_LINE("Enter file with input data");
  GET_LINE(FILE_NAME,NAME_LENGTH);
  OPEN(INF, MODE => IN_FILE, NAME =>
  FILE_NAME(1..NAME_LENGTH));
  GET_LINE(INF,TITLE,TITLE_LENGTH);
  PUT_LINE(TITLE(1..TITLE_LENGTH));
  while not END_OF_FILE(INF) loop
    GET(INF,MY_STRING);
    if MY_STRING = "NODE" then --insert a node
      GET(INF,MY_NODE);
      GET(INF,MY_EXE);
      INSERT_NODE(G, MY_NODE, CONVERT(MY_EXE), SUCCESS);
    else -- insert an edge
      GET(INF,MY_FROM);
      GET(INF,MY_TO);
      GET(INF,MY_COST);
      INSERT_EDGE(G, MY_FROM, MY_TO, CONVERT(MY_COST),
      SUCCESS);
    end if;
    SKIP_LINE(INF);
  end loop;

  -- search and print elements of the created graph
  PUT_LINE("Breadth First Search");
  BREADTH_FIRST_SEARCH(G);
  NUM_OF_TASKS := AUX_COUNT;
  MAX_INDEX := (NUM_OF_TASKS*(NUM_OF_TASKS-1))/2;
  NEW_LINE;
  PUT_LINE("Number of Tasks = ");
  PUT(NUM_OF_TASKS, WIDTH => 3);
```

```
   NEW_LINE;
   UNI_EXE_COST := AUX_COST;
   -- print uniprocessor execution time
   PUT_LINE("Uniprocessor Execution Time");
   PUT(UNI_EXE_COST, FORE => 4, AFT => 4, EXP => 0);
   NEW_LINE;

   PUT_LINE("Depth First Search");
   DEPTH_FIRST_SEARCH(G);

   --find the longest path
   FIND_LONG_PATH(G,OUT_SET);
   PUT_LINE("LONG_PATH");
   PRINT_SET(OUT_SET);



   -- initializes parent array
   PUT_LINE("Parent array calculation");
   for J in 2..NUM_OF_TASKS loop
     for I in 1..J-1 loop
       RETRIEVE_EDGE(G, I, J, COMIJ, SUCCESS);
       if SUCCESS then
         MY_DIS.INSERT(MY_ATOM(I),PARENT_ARRAY(MY_ATOM(J)));
       end if;
     end loop;
     PRINT_SET(PARENT_ARRAY(MY_ATOM(J)));
   end loop;


   --initializes ancestor array
   PUT_LINE("Ancestor array calculation");
   for J in 2..NUM_OF_TASKS loop
     for I in 1..J-1 loop
       RETRIEVE_EDGE(G, I, J, COMIJ, SUCCESS);
       if SUCCESS then
         MY_DIS.INSERT(MY_ATOM(I),ANCESTOR_ARRAY(MY_ATOM(J)));
         if I /= 1 then
           ANCESTOR_ARRAY(MY_ATOM(J)) :=
           MY_DIS.UNION(ANCESTOR_ARRAY(MY_ATOM(J)),
           ANCESTOR_ARRAY(MY_ATOM(I)));
         end if;
       end if;
     end loop;
     PRINT_SET(ANCESTOR_ARRAY(MY_ATOM(J)));
   end loop;

end CONSTRUCT_TASK_FLOW;
```

C. CALHEU.ADA

```
separate(STATICAL)

procedure CALC_HEURISTIC(G: in out DGRAPH;
                          TEMP_ARRAY: in out SORT_ARRAY) is

 SUCCESS: BOOLEAN;
 SUCCESS1, SUCCESS2: BOOLEAN;
 VALUE: INTEGER;
 type H_FUNCTION is array(1..MAX_PROCESS, 1..MAX_PROCESS) of
 FLOAT;
 H: H_FUNCTION := (others => (others => 0.0));
 CIJ, EI, EJ: MY_ELEMENT;
 COUNT: INTEGER := 0;
 IS_MEMBER: BOOLEAN;
 AUX_TERM: FLOAT;

 function HEURISTIC1(EXI, EXJ, COMIJ: MY_ELEMENT) return
 FLOAT is

 begin
   return (0.5*COMIJ.INFO + (0.5/(EXI.INFO+EXJ.INFO)));
 end HEURISTIC1;

 procedure PRINT_HEADER is

 begin
   TEXT_IO.PUT_LINE("I    J     EI     EJ      CIJ       HIJ");
 end PRINT_HEADER;


begin
     for I in 1..NUM_OF_TASKS-1 loop
       for J in I+1..NUM_OF_TASKS loop
       COUNT := COUNT + 1;
       MY_DIS.INSERT(MY_ATOM(I), TEMP_ARRAY(COUNT).PAIR_SET);
       TEMP_ARRAY(COUNT).INDEX_FROM := I;
       MY_DIS.INSERT(MY_ATOM(J), TEMP_ARRAY(COUNT).PAIR_SET);
       TEMP_ARRAY(COUNT).INDEX_TO := J;
       RETRIEVE_NODE(G, I, EI, SUCCESS1);
       if not SUCCESS1 then
         EI.INFO := 0.0;
       end if;
       RETRIEVE_NODE(G, J, EJ, SUCCESS2);
       if not SUCCESS2 then
```

166

```
          EJ.INFO := 0.0;
        end if;
        RETRIEVE_EDGE(G, I, J, CIJ, SUCCESS);
        AUX_TERM := 0.0;
        if not SUCCESS then
          CIJ.INFO   := 0.0;
          IS_MEMBER := MY_DIS.MEMBER(MY_ATOM(I),
          ANCESTOR_ARRAY(MY_ATOM(J)));
          if IS_MEMBER then
            AUX_TERM := 1.0;
          end if;
        end if;
        if SUCCESS1 and SUCCESS2 then
          H(I,J) := HEURISTIC1(EI, EJ, CIJ);
        else
          H(I,J) := 0.0;
        end if;

        TEMP_ARRAY(COUNT).EFROM := EI.INFO;
        TEMP_ARRAY(COUNT).ETO   := EJ.INFO;
        TEMP_ARRAY(COUNT).COMM  := CIJ.INFO;
        TEMP_ARRAY(COUNT).HINFO := H(I,J);
      end loop;
    end loop;


    NEW_SORT.QS_3(TEMP_ARRAY(1..MAX_INDEX));
    PRINT_HEADER;
    for I in 1..MAX_INDEX loop
      PUT(TEMP_ARRAY(I).INDEX_FROM, WIDTH => 2);
      PUT("   ");
      PUT(TEMP_ARRAY(I).INDEX_TO, WIDTH => 2);
      PUT("   ");
      PUT(TEMP_ARRAY(I).EFROM, FORE => 2, AFT => 3, EXP => 0);
      PUT(" ");
      PUT(TEMP_ARRAY(I).ETO, FORE => 2, AFT => 3, EXP => 0);
      PUT(" ");
      PUT(TEMP_ARRAY(I).COMM, FORE => 2, AFT => 3, EXP => 0);
      PUT(" ");
      PUT(TEMP_ARRAY(I).HINFO, FORE => 2, AFT => 3, EXP => 0);
      NEW_LINE;
    end loop;
end CALC_HEURISTIC;
```

D. ALLOC.ADA

```
separate(STATICAL)

procedure ALLOCATE(G: in out DGRAPH; LAST_PROC: out
                   PROCESSOR;
                     TEMP_ARRAY: in out SORT_ARRAY;
                     TEMP_SET: in out SET; PROC_ARRAY: in out
                   PROCESSOR_ARRAY;
                     UTIL_ARRAY: in out UTILIZATION_ARRAY;
                     ALLOCATED_TASKS: in out SET) is


  L_PROC: PROCESSOR;
  TOP_DOWN_INDEX: INTEGER;
  BOTTOM_UP_INDEX: INTEGER := 1;
  PAIR_TOGETHER, PAIR_SEPARATED: SET;
  ALLOCATED: BOOLEAN;
  SET_TO_ALLOC: SET;
  BOTH: BOOLEAN;
  PROC_USED: PROCESSOR;
  CURRENT_ATOM: MY_ATOM;
  CURRENT_SET: SET;
  CURRENT_PROCESSOR: PROCESSOR;
  VAR_UTILIZATION: FLOAT;

  function FIND_LEAST_USED_PROCESSOR(UTIL: UTILIZATION_ARRAY)

    return PROCESSOR is

    LUP: PROCESSOR := PROCESSOR'FIRST;

  begin
    for I in PROCESSOR'SUCC(PROCESSOR'FIRST)..L_PROC loop
      if UTIL(I) < UTIL(LUP) then
        LUP := I;
      end if;
    end loop;
    return LUP;
  end FIND_LEAST_USED_PROCESSOR;

  function FIND_LEAST_USED_PROCESSOR(UTIL: UTILIZATION_ARRAY;
                                     PROC: PROCESSOR)
    return PROCESSOR is

  LUP: PROCESSOR;
  FIRST: BOOLEAN := TRUE;

  begin
```

```
   for I in PROCESSOR'FIRST..L_PROC loop
     if I /= PROC then
       if FIRST then
         LUP := I;
         FIRST := FALSE;
       else
         if UTIL(I) < UTIL(LUP) then
           LUP := I;
         end if;
       end if;
     end if;
   end loop;
   return LUP;
 end FIND_LEAST_USED_PROCESSOR;


 procedure ADDITIONAL_UTILIZATION(ADDED_TASKS: in out SET;
ACCUM: out FLOAT) is

 ACC: FLOAT := 0.0;
 THE_KEY: MY_ATOM;
 THE_ELEMENT: MY_ELEMENT;
 OK: BOOLEAN := TRUE;
 SUCCESS: BOOLEAN;

 begin

    MY_DIS.TAKE_OUT_MEMBER(ADDED_TASKS, THE_KEY, OK);
    while OK loop
      RETRIEVE_NODE(G, INTEGER(THE_KEY), THE_ELEMENT,
      SUCCESS);
      ACC := ACC + THE_ELEMENT.INFO;
      MY_DIS.TAKE_OUT_MEMBER(ADDED_TASKS, THE_KEY, OK);
    end loop;
    ACCUM := ACC;
 end ADDITIONAL_UTILIZATION;

 procedure ALLOCATE_SET_OF_TASKS_TO_PROCESSOR(SET_OF_TASKS:
                                 in SET;
                                 PROCESSOR_DEST: in
                                 PROCESSOR;
                                 ALLOCATED_TASKS: in out
                                 SET;
                                 DELTA_UTILIZATION: in out
                                 FLOAT;
                                 U_ARRAY: in out
                                 UTILIZATION_ARRAY;
                                 P_ARRAY: in out
                                 PROCESSOR_ARRAY) is

 ADDED_TASKS: SET;
```

169

```
begin
  ADDED_TASKS := MY_DIS.DIFFERENCE(SET_OF_TASKS,
  ALLOCATED_TASKS);
  ALLOCATED_TASKS := MY_DIS.UNION(SET_OF_TASKS,
  ALLOCATED_TASKS);
  P_ARRAY(PROCESSOR_DEST) := MY_DIS.UNION(SET_OF_TASKS,
  P_ARRAY(PROCESSOR_DEST));
  ADDITIONAL_UTILIZATION(ADDED_TASKS, DELTA_UTILIZATION);
  U_ARRAY(PROCESSOR_DEST) := U_ARRAY(PROCESSOR_DEST) +
  DELTA_UTILIZATION;
end ALLOCATE_SET_OF_TASKS_TO_PROCESSOR;


procedure IS_PAIR_ALLOCATED (PAIR: in SET;
                             ANSWER: out BOOLEAN;
                             TO_BE_ALLOC: out SET;
                             ALLOCATE_BOTH:out BOOLEAN;
                             ALREADY_USED: out PROCESSOR)
is

AUX: SET;
USED: INTEGER;
AUX_PROCESSOR: PROCESSOR := P1;
AUX_ATOM: MY_ATOM;
OK:  BOOLEAN;

begin
  AUX := MY_DIS.INTERSECTION(PAIR, ALLOCATED_TASKS);
  USED := MY_DIS.COUNT_MEMBERS(AUX);
  if USED = 2 then
    ANSWER := TRUE;
  else
    ANSWER := FALSE;
    TO_BE_ALLOC := MY_DIS.DIFFERENCE(PAIR, AUX);
    if USED = 1 then
      MY_DIS.TAKE_OUT_MEMBER(AUX, AUX_ATOM, OK);
      FIND_PROCESSOR(PROC_ARRAY, AUX_ATOM, L_PROC, OK,
      AUX_PROCESSOR);
      ALREADY_USED := AUX_PROCESSOR;
      ALLOCATE_BOTH := FALSE;
    else
      ALLOCATE_BOTH := TRUE;
      ALREADY_USED := AUX_PROCESSOR;
    end if;
  end if;
end IS_PAIR_ALLOCATED;


procedure ALLOCATE_PAIR_SEPARATE is
```

```
SUCCESS: BOOLEAN;

begin
  -- allocatte separate tasks
  IS_PAIR_ALLOCATED(PAIR_SEPARATED, ALLOCATED, SET_TO_ALLOC,
  BOTH, PROC_USED);
  if not ALLOCATED then -- there is at least one task not
    -- allocated
    if not BOTH then -- only one must be allocated
      CURRENT_PROCESSOR :=
      FIND_LEAST_USED_PROCESSOR(UTIL_ARRAY, PROC_USED);
      ALLOCATE_SET_OF_TASKS_TO_PROCESSOR(SET_TO_ALLOC,
                                         CURRENT_PROCESSOR,
                                         ALLOCATED_TASKS,
                                         VAR_UTILIZATION,
                                         UTIL_ARRAY,
                                         PROC_ARRAY);

    else
      CURRENT_PROCESSOR :=
      FIND_LEAST_USED_PROCESSOR(UTIL_ARRAY);
      CURRENT_SET := SET_TO_ALLOC;
      MY_DIS.TAKE_OUT_MEMBER(SET_TO_ALLOC, CURRENT_ATOM,
      SUCCESS);
      ALLOCATE_SET_OF_TASKS_TO_PROCESSOR(SET_TO_ALLOC,
                                         CURRENT_PROCESSOR,
                                         ALLOCATED_TASKS,
                                         VAR_UTILIZATION,
                                         UTIL_ARRAY,
                                         PROC_ARRAY);
      SET_TO_ALLOC := MY_DIS.DIFFERENCE(CURRENT_SET,
      SET_TO_ALLOC);
      CURRENT_PROCESSOR :=
      FIND_LEAST_USED_PROCESSOR(UTIL_ARRAY);
      ALLOCATE_SET_OF_TASKS_TO_PROCESSOR(SET_TO_ALLOC,
                                         CURRENT_PROCESSOR,
                                         ALLOCATED_TASKS,
                                         VAR_UTILIZATION,
                                         UTIL_ARRAY,
                                         PROC_ARRAY);

    end if;
  end if;
end ALLOCATE_PAIR_SEPARATE;


procedure ALLOCATE_PAIR_TOGETHER is

begin
  -- allocate tasks that should be together
  IS_PAIR_ALLOCATED(PAIR_TOGETHER, ALLOCATED, SET_TO_ALLOC,
```

```
                        BOTH, PROC_USED);
   if not ALLOCATED then -- at least one should be allocated
     if not BOTH then     -- allocate only one task
       if PROC_USED = P1 then
           CURRENT_PROCESSOR :=
           FIND_LEAST_USED_PROCESSOR(UTIL_ARRAY, P1);
           ALLOCATE_SET_OF_TASKS_TO_PROCESSOR(SET_TO_ALLOC,
                                       CURRENT_PROCESSOR,
                                       ALLOCATED_TASKS,
                                       VAR_UTILIZATION,
                                       UTIL_ARRAY,
                                       PROC_ARRAY);
       else
         ALLOCATE_SET_OF_TASKS_TO_PROCESSOR(SET_TO_ALLOC,
                                       PROC_USED,
                                       ALLOCATED_TASKS,
                                       VAR_UTILIZATION,
                                       UTIL_ARRAY,
                                       PROC_ARRAY);
       end if;
     else
       CURRENT_PROCESSOR :=
       FIND_LEAST_USED_PROCESSOR(UTIL_ARRAY);
       ALLOCATE_SET_OF_TASKS_TO_PROCESSOR(SET_TO_ALLOC,
                                       CURRENT_PROCESSOR,
                                       ALLOCATED_TASKS,
                                       VAR_UTILIZATION,
                                       UTIL_ARRAY,
                                       PROC_ARRAY);
     end if;
   end if;

end ALLOCATE_PAIR_TOGETHER;

-- get the options from keyboard
procedure GET_OPTIONS(L_PROC: out PROCESSOR) is

procedure QUERY_NR_OF_PROCESSORS(P: out PROCESSOR) is

NUM_PROC:INTEGER;

begin
  PUT_LINE("Please, enter number of processors?");
  GET(NUM_PROC);
  case NUM_PROC is
    when 1 =>
      P := P1;
    when 2 =>
      P := P2;
    when 3 =>
      P := P3;
```

```
      when 4 =>
        P := P4;
      when 5 =>
        P := P5;
      when 6 =>
        P := P6;
      when 7 =>
        P := P7;
      when 8 =>
        P := P8;
      when 9 =>
        P := P9;
      when 10 =>
        P := P10;
      when others =>
        P := P10;
    end case;
end QUERY_NR_OF_PROCESSORS;

begin
 QUERY_NR_OF_PROCESSORS(L_PROC);
end GET_OPTIONS;


 begin

   TOP_DOWN_INDEX := MAX_INDEX;
   -- starts the allocation
   CURRENT_PROCESSOR :=
   FIND_LEAST_USED_PROCESSOR(UTIL_ARRAY);
   ALLOCATE_SET_OF_TASKS_TO_PROCESSOR(RESULT_SET,
   CURRENT_PROCESSOR, ALLOCATED_TASKS, VAR_UTILIZATION,
   UTIL_ARRAY, PROC_ARRAY);

   -- print potential parallel execution time
   PUT_LINE("Potential Parallel Execution Time:");
   PUT(VAR_UTILIZATION, FORE => 4, AFT => 4, EXP => 0);
   NEW_LINE;

   -- print potential speed-up
   POTENTIAL_SPEED_UP := UNI_EXE_COST/VAR_UTILIZATION;
   PUT_LINE("Potential Speed-Up: ");
   PUT(POTENTIAL_SPEED_UP, FORE => 4, AFT => 4, EXP => 0);
   NEW_LINE;

   GET_OPTIONS(L_PROC);

   PUT_LINE(" bottom_up top_down ");
   while (not MY_DIS.IS_FULL(ALLOCATED_TASKS)) and then
     (BOTTOM_UP_INDEX <= TOP_DOWN_INDEX) loop
     PUT(BOTTOM_UP_INDEX, WIDTH => 6);
```

```
          PUT("     ");
          PUT(TOP_DOWN_INDEX, WIDTH => 6);
          NEW_LINE;
          PAIR_SEPARATED := TEMP_ARRAY(BOTTOM_UP_INDEX).PAIR_SET;
          BOTTOM_UP_INDEX := BOTTOM_UP_INDEX + 1;
          PAIR_TOGETHER := TEMP_ARRAY(TOP_DOWN_INDEX).PAIR_SET;
          TOP_DOWN_INDEX := TOP_DOWN_INDEX - 1;
          ALLOCATE_PAIR_SEPARATE;
          ALLOCATE_PAIR_TOGETHER;
       end loop;

       for I in PROCESSOR'FIRST..L_PROC loop
          PRINT_PROCESSOR(I, L_PROC);
          PRINT_SET(PROC_ARRAY(I));
          PUT(UTIL_ARRAY(I), FORE => 4, AFT => 4, EXP => 0);
          NEW_LINE;
       end loop;

       LAST_PROC := L_PROC;

    end ALLOCATE;


    E.   SCHED.ADA


    separate(STATICAL)

    procedure SCHEDULE(G: in out DGRAPH;
                       L_PROC: in PROCESSOR;
                       P: in out PROCESSOR_ARRAY;
                       S: in out SCHEDULE_ARRAY;
                       T: out TIME_ARRAY) is

    MY_P: PROCESSOR_ARRAY := P;
    MY_T: TIME_ARRAY := (others => 0.0);

    procedure SCHEDULE_TASKS(G: in out DGRAPH; P: in
    PROCESSOR_ARRAY;
                             S: in out SCHEDULE_ARRAY) is

    MY_SCHEDULE: SCHEDULE_ARRAY;
    SUCCESS: BOOLEAN;
    CURRENT_PROCESSOR, SOURCE_PROCESSOR: PROCESSOR;
    CURRENT_ELEMENT, EDGE_ELEMENT: MY_ELEMENT;
    --type TIME_ARRAY is array(PROCESSOR) of FLOAT;
    --MY_TIME: TIME_ARRAY := (others => 0.0);
    MY_INFO: EXECUTION_INFO;
    AUX_INFO: EXECUTION_INFO;
    type AUX_ARRAY is array(MY_ATOM) of FLOAT;
```

174

```ada
EXTRA_ARRAY: AUX_ARRAY := (others => 0.0);
FOUND: BOOLEAN := FALSE;


function MAXIMUM(X,Y,W: FLOAT) return FLOAT is

Z: FLOAT := X;

begin
   if Y > Z then
     Z := Y;
   end if;
   if W > Z then
     Z := W;
   end if;
   return Z;
end MAXIMUM;

begin
   for I in MY_ATOM'FIRST..MY_ATOM'LAST loop
     FIND_PROCESSOR(P, I, L_PROC, SUCCESS,
     CURRENT_PROCESSOR);
     RETRIEVE_NODE(G, INTEGER(I), CURRENT_ELEMENT, SUCCESS);
     if I = 1 then
       MY_INFO.START_TIME := MY_T(CURRENT_PROCESSOR);
       MY_INFO.END_TIME   := MY_INFO.START_TIME +
       CURRENT_ELEMENT.INFO;
       MY_INFO.TASK_ID    := I;
       MY_INFO.IS_COMM    := FALSE;
       MY_T(CURRENT_PROCESSOR) := MY_INFO.END_TIME;
       MY_SCHEDULE(MY_ATOM(I)) := MY_INFO;
     else
       for J in MY_ATOM'FIRST..MY_ATOM'PRED(I) loop
         if I /= J then
           RETRIEVE_EDGE(G, INTEGER(J), INTEGER(I),
           EDGE_ELEMENT, SUCCESS);
           if SUCCESS then
             AUX_INFO := MY_SCHEDULE(MY_ATOM(J));
             FOUND := FALSE;
             MY_INFO.START_TIME := MAXIMUM(AUX_INFO.END_TIME,
                                   EXTRA_ARRAY(I),
                                   MY_T(CURRENT_PROCESSOR));
             EXTRA_ARRAY(I) := MY_INFO.START_TIME;
           end if;
         end if;
       end loop;

       MY_INFO.END_TIME := MY_INFO.START_TIME +
       CURRENT_ELEMENT.INFO;
       MY_INFO.TASK_ID := I;
       MY_INFO.IS_COMM := FALSE;
```

```
        MY_T(CURRENT_PROCESSOR) := MY_INFO.END_TIME;
        MY_SCHEDULE(MY_ATOM(I)) := MY_INFO;
      end if;
   end loop;
  S := MY_SCHEDULE;
end SCHEDULE_TASKS;


 procedure PRINT_SCHEDULE(S: in out SCHEDULE_ARRAY;
                          P: in out PROCESSOR_ARRAY) is

 FOUND: BOOLEAN := FALSE;
 SUCCESS: BOOLEAN := TRUE;
 CURRENT_ATOM: MY_ATOM;
 CURRENT_INFO: EXECUTION_INFO;

 procedure PRINT_P(PROC: in PROCESSOR) is

 AUX_INDEX: INTEGER := 0;

 begin
    for I in PROCESSOR'FIRST..L_PROC loop
      AUX_INDEX := AUX_INDEX+1;
      if I = PROC then
        PUT("PROCESSOR P");
        PUT(AUX_INDEX, WIDTH => 1);
        NEW_LINE;
        exit;
      end if;
   end loop;

 PUT_LINE("START        END     TASK");

 end PRINT_P;

 begin
  for I in PROCESSOR'FIRST..L_PROC loop
    PRINT_P(I);
    MY_DIS.TAKE_OUT_MEMBER(MY_P(I),CURRENT_ATOM,SUCCESS);
    while SUCCESS loop
      CURRENT_INFO := S(CURRENT_ATOM);
      PUT(CURRENT_INFO.START_TIME, FORE => 4, AFT => 4, EXP
        => 0);
       PUT("   ");
       PUT(CURRENT_INFO.END_TIME, FORE => 4, AFT => 4, EXP
        => 0);
       PUT("   ");
       PUT(INTEGER(CURRENT_INFO.TASK_ID), WIDTH => 2);
       NEW_LINE;
       MY_DIS.TAKE_OUT_MEMBER(MY_P(I),CURRENT_ATOM,SUCCESS);
    end loop;
```

```
  end loop;

 end PRINT_SCHEDULE;

begin
  SCHEDULE_TASKS(G,P,S);
  PRINT_SCHEDULE(S,P);
  T := MY_T;
end SCHEDULE;
```

F. IMPROVE.ADA

```
 separate(STATICAL)

 procedure IMPROVE is


--Data for improvement of the task allocation by pairwise
exchange of tasks
CURR_P_ARRAY: PROCESSOR_ARRAY := (others => MY_DIS.CREATE);
CURR_THE_SCHEDULE: SCHEDULE_ARRAY;
CURR_MY_TIME: TIME_ARRAY;
COST_ARRAY: TIME_ARRAY;
CURR_COST_ARRAY: TIME_ARRAY;
TOT_COST: TIME_ARRAY;
CURR_TOT_COST: TIME_ARRAY;
type TOPOLOGY_COST_ARRAY is array(P1..P4,P1..P4) of FLOAT;
HOP_ARRAY: TOPOLOGY_COST_ARRAY := ((0.0,1.0,2.0,1.0),
                                   (1.0,0.0,1.0,2.0),
                                   (2.0,1.0,0.0,1.0),
                                   (1.0,2.0,1.0,0.0));


SET_COUNT, INDICATOR: INTEGER;
GUESS: FLOAT;
CONTROL_SET: SET := MY_DIS.CREATE;
AUX_SET    : SET := MY_DIS.CREATE;
type CHANGE_ARRAY is array(MY_ATOM) of MY_ATOM;
SWAP_ARRAY: CHANGE_ARRAY;
THIS_ATOM: MY_ATOM;
SUCCESS:   BOOLEAN;
PROC1, PROC2: PROCESSOR;
MAX_COST: FLOAT;
CURR_MAX_COST: FLOAT;
```

177

```
-- Calculates cost array
procedure CALC_COST_ARRAY(G: in out DGRAPH;
                          P: in out PROCESSOR_ARRAY;
                          C_ARRAY: out TIME_ARRAY)  is

TEMP_COST_ARRAY: TIME_ARRAY := (others => 0.0);
COST_ELEMENT: MY_ELEMENT;
PROD_COST: FLOAT;
SUCCESS: BOOLEAN;
PROC_FROM, PROC_TO: PROCESSOR;

begin
   for I in 1..NUM_OF_TASKS loop
     for J in 1..NUM_OF_TASKS loop

        if I /= J then
          RETRIEVE_EDGE(G, I, J, COST_ELEMENT, SUCCESS);
          if SUCCESS then
            FIND_PROCESSOR(P,MY_ATOM(I),P4,SUCCESS,PROC_FROM);
            FIND_PROCESSOR(P,MY_ATOM(J),P4,SUCCESS,PROC_TO);
            if PROC_FROM /= PROC_TO then -- if they are in
              dif. processors
              PROD_COST :=
              HOP_ARRAY(PROC_FROM,PROC_TO)*COST_ELEMENT.INFO;
              TEMP_COST_ARRAY(PROC_FROM) :=
              TEMP_COST_ARRAY(PROC_FROM) +
              PROD_COST;
              TEMP_COST_ARRAY(PROC_TO) :=
              TEMP_COST_ARRAY(PROC_TO)   +
              PROD_COST;
            end if;
          end if;
        end if;
     end loop;
   end loop;
   C_ARRAY := TEMP_COST_ARRAY;

end CALC_COST_ARRAY;

function FIND_MAX_COST(T_ARRAY: in TIME_ARRAY)
return FLOAT is

MAX: FLOAT := T_ARRAY(P1);

begin
   for I in P2..P4 loop
     if T_ARRAY(I) > MAX then
       MAX := T_ARRAY(I);
     end if;
   end loop;
   return MAX;
```

178

```
      end FIND_MAX_COST;

      procedure PRINT_NODE_PROCESSOR(P: in PROCESSOR) is

      begin
        case P is
          when P1 =>
            PUT("P1");
          when P2 =>
            PUT("P2");
          when P3 =>
            PUT("P3");
          when P4 =>
            PUT("P4");
          when others =>
            null;
        end case;
      end PRINT_NODE_PROCESSOR;

      procedure PRINT_TOT_COST is

      begin
        PUT_LINE ("PROCESSOR       MY_TIME      COST_ARRAY
        TOTAL_COST");
          for I in P1..P4 loop
            TOT_COST(I) := MY_TIME(I) + COST_ARRAY(I);
            PRINT_NODE_PROCESSOR(I);
            PUT("           ");
            PUT(MY_TIME(I), FORE => 4, AFT => 4, EXP => 0);
            PUT("       ");
            PUT(COST_ARRAY(I), FORE => 4, AFT => 4, EXP => 0);
            PUT("     ");
            PUT(TOT_COST(I), FORE => 4, AFT => 4, EXP => 0);
            NEW_LINE;
          end loop;
      end PRINT_TOT_COST;


      begin
        CALC_COST_ARRAY(DG,P_ARRAY,COST_ARRAY);
        PRINT_TOT_COST;
        MAX_COST := FIND_MAX_COST(TOT_COST);


        -- starts allocation improvement
        RANDOM.SET_SEED;
        for K in 1..100 loop
          for I in 1..NUM_OF_TASKS loop
            MY_DIS.INSERT(MY_ATOM(I),CONTROL_SET);
          end loop;
          SET_COUNT := NUM_OF_TASKS;
```

179

```
for I in 1..NUM_OF_TASKS loop
  GUESS := RANDOM.SRAND;
  for J in 1..SET_COUNT loop
    if GUESS < (FLOAT(J)/FLOAT(SET_COUNT)) then
      INDICATOR := J;
      exit;
    end if;
  end loop;
  for J in 1..SET_COUNT loop
    MY_DIS.TAKE_OUT_MEMBER
    (CONTROL_SET,THIS_ATOM,SUCCESS);
    if J = INDICATOR then
      SWAP_ARRAY(MY_ATOM(I)) := THIS_ATOM;
    else
      MY_DIS.INSERT(THIS_ATOM,AUX_SET);
    end if;
  end loop;
  CONTROL_SET := AUX_SET;
  SET_COUNT := SET_COUNT - 1;
  MY_DIS.CLEAR_SET(AUX_SET);
end loop;

for I in 1..NUM_OF_TASKS/2 loop
  FIND_PROCESSOR(P_ARRAY,SWAP_ARRAY(MY_ATOM(2*I-1)),
  P4,SUCCESS,PROC1);
  FIND_PROCESSOR(P_ARRAY,SWAP_ARRAY(MY_ATOM(2*I)),
  P4,SUCCESS,PROC2);
  if (PROC1 = P1 or PROC2 = P1) then
    MY_DIS.INSERT(SWAP_ARRAY(MY_ATOM(2*I-1)),
    CURR_P_ARRAY(PROC1));
    MY_DIS.INSERT(SWAP_ARRAY(MY_ATOM(2*I)),
    CURR_P_ARRAY(PROC2));
  else
    if PROC1 = PROC2 then
      MY_DIS.INSERT(SWAP_ARRAY(MY_ATOM(2*I-1)),
      CURR_P_ARRAY(PROC1));
      MY_DIS.INSERT(SWAP_ARRAY(MY_ATOM(2*I)),
      CURR_P_ARRAY(PROC1));
    else
      MY_DIS.INSERT(SWAP_ARRAY(MY_ATOM(2*I-1)),
      CURR_P_ARRAY(PROC2));
      MY_DIS.INSERT(SWAP_ARRAY(MY_ATOM(2*I)),
      CURR_P_ARRAY(PROC1));
    end if;
  end if;
end loop;
if NUM_OF_TASKS mod 2 = 1 then
  FIND_PROCESSOR(P_ARRAY,SWAP_ARRAY
  (MY_ATOM(NUM_OF_TASKS)),P4,SUCCESS,PROC1);
  MY_DIS.INSERT(SWAP_ARRAY
  (MY_ATOM(NUM_OF_TASKS)),CURR_P_ARRAY(PROC1));
```

```
      end if;

   PUT_LINE("ITERATION");
   PUT(K, WIDTH => 3);
   NEW_LINE;

   for I in P1..P4 loop
     PRINT_NODE_PROCESSOR(I);
     PUT("     ");
     PRINT_SET(CURR_P_ARRAY(I));
   end loop;

   SCHEDULE(DG,LAST_PROCESSOR,CURR_P_ARRAY,
   CURR_THE_SCHEDULE,CURR_MY_TIME);
   CALC_COST_ARRAY(DG,CURR_P_ARRAY,
   CURR_COST_ARRAY);

   for I in P1..P4 loop
     CURR_TOT_COST(I)  := CURR_MY_TIME(I) +
     CURR_COST_ARRAY(I);
   end loop;
   CURR_MAX_COST := FIND_MAX_COST(CURR_TOT_COST);

   PUT_LINE("CURR MAX COST");
   PUT(CURR_MAX_COST, FORE => 4, AFT => 4, EXP => 0);
   NEW_LINE;
   PUT_LINE("MAX COST");
   PUT(MAX_COST, FORE => 4, AFT => 4, EXP => 0);
   NEW_LINE;


   if CURR_MAX_COST < MAX_COST then
     P_ARRAY := CURR_P_ARRAY;
     MY_TIME := CURR_MY_TIME;
     COST_ARRAY := CURR_COST_ARRAY;
     TOT_COST := CURR_TOT_COST;
     MAX_COST := CURR_MAX_COST;
     THE_SCHEDULE := CURR_THE_SCHEDULE;
   end if;



   for I in P1..P4 loop
     MY_DIS.CLEAR_SET(CURR_P_ARRAY(I));
   end loop;

end loop;

PUT_LINE("IMPROVED ALLOCATION");
for I in P1..P4 loop
  PRINT_NODE_PROCESSOR(I);
```

```
      PUT("    ");
      PRINT_SET(P_ARRAY(I));
    end loop;

    PRINT_TOT_COST;

  end IMPROVE;
```

LIST OF REFERENCES

[ALSYS 90]
Alsys Inc."PC Mothered Transputer Cross Compilation User Manuals", Alsys, Burlington, MA, May 1990.

[ATKINSON 88]
Atkinson C., Moreton T., Natali A., "Ada for Distributed Systems", Cambridge University Press, 1988.

[DJKSTRA 59]
Djkstra, E.W., A *Note on two Problems in Connexion with Graphs,* Numerical Mathematics vol.1, Oct. 1959.

[HOARE 78]
Hoare, C. A. R., *Communicating Sequential Processes,* Communications of the ACM, vol. 21, no. 8, Aug. 1978.

[QUINN 87]
Quinn, M. J., "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill, 1987.

[RAMAMRITHAM 89]
Ramamrithan, K., *Allocating and Scheduling of Complex Periodic Tasks,* University of Massachusetts, Amherst, Technical Report 90-01, Oct. 1989.

[REYNOLDS 83]
Reynolds, P. F., *The Implementation and Use of Ada on a Distributed System with High Reliability Requirements,* Final Report on NASA grant number:NAG-1-260,University of Virginia, 1983.

[RICHMOND 91]
Richmond, C., *On Programming Transputers to Capture Ada Multitasking for the NPS Autonomous Underwater Vehicle,* Master's thesis, Naval Postgraduate School, Monterey, CA, December 1991.

[STANKOVIC 87]
Stankovic, K. A., Ramamritham K., Cheng S., *Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey,* University of Massachusetts, Jul. 1987.

[STANKOVIC 88]
Stankovic, K. A., *Real-Time Computing Systems: The Next Generation,* University of Massachusetts, Feb. 1988.

[STUBBS 87]
    Stubbs, D., Webre, N., "Data Structures with Abstract
    Data   Types and Modula-2", Brooks/Cole, 1987.

[TANENBAUN 89]
    Tanenbaum, A. S., "Computer Networks", Prentice Hall,
    1989.

[TSUCHIYA 82]
    Tsuchiya, M., Ma, P. R., Lee, E. Y. S., *A Task
    Allocation Model for Distributed Computing Systems*, IEEE
    Transactions on Computers vol. 31, No. 1, 1982.

[TZENG 91]
    Tzeng, N., *Enhanced Hypercubes*, IEEE Transactions on
    Computers, vol. 40, No. 3, March 1991.

[VOLZ 85]
    Volz, R. A., *Some Problems in Distributing Real-time
    Ada Programs Across Machines*, Ada in Use, Proceedings
   of the Ada International Conference, Paris, 1985.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center   2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 52   2
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Chairman, Code EC   1
   Department of Electrical & Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5002

4. Professor Shridhar Shukla, Code EC/Sh   2
   Department of Electrical & Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5002

5. Professor Robert B. McGhee, Code CS/Mz   1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5002

6. Professor Uno R. Kodres, Code CS/Kr   1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5002

7. Professor Amr M. Zaky, Code CS/Za   1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5002

8. Professor Anthony Healey, Code ME/Hy   1
   Department of Mechanical Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5002

9. Diretoria de Armamento e Comunicaçoes da Marinha   2
   Brazilian Navy
   Rua Primeiro de Março, 118, 21° andar
   CEP 20010, Rio de Janeiro, RJ, Brasil

10. LCdr Marco A. G. Falcao
    Brazilian Navy
    Rua Primeiro de Março, 118, 21° andar
    CEP 20010, Rio de Janeiro, RJ, Brasil

1