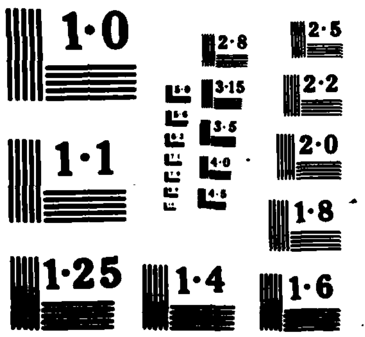
AD-A171 098 PARTITIONING REAL-TIME ADA (TRADE NAME) SOFTWARE FOR DISTRIBUTED TARGETS(U) GTE LABS INC WALTHAM MA COMPUTER SCIENCE LAB A CHOW ET AL. MAY 86 TN-86-058.01 1/1

UNCLASSIFIED N00014-85-C-0796 F/G 9/2 NL


1·0
2·8
2·5
3·15
2·2
3·5
1·1
4·0
2·0
4·5
1·8
1·25
1·4
1·6

12

Technical Note No. 86-858.01

PARTITIONING REAL-TIME ADA®[1] SOFTWARE FOR DISTRIBUTED TARGETS[2]

by

A. Chow
M. Feridun

May 1986

Computer Science Laboratory
GTE LABORATORIES INCORPORATED
40 Sylvan Road
Waltham, Massachusetts 02254

[redacted]

[1]Ada is a registered trademark of the U.S. Department of Defense, A.J.P.O.

[2]This research has been partially supported by the Office of Naval Research under grant number N00014- 85-C-0796.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ADA 171098

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 86-858.01	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Partitioning Real-Time Ada Software for Distributed Targets		5. TYPE OF REPORT & PERIOD COVERED Technical Note
		6. PERFORMING ORG. REPORT NUMBER 86-858.01
7. AUTHOR(s) A. Chow M. Feridun		8. CONTRACT OR GRANT NUMBER(s) N00014-85-C-0796
9. PERFORMING ORGANIZATION NAME AND ADDRESS GTE Laboratories Incorporated Computer Science Laboratory 40 Sylvan Road, Waltham, MA 02254		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N, RR014-08, RR014-08-01, NR 049-635
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research (Code 1133) 800 N. Quincy Street Arlington, VA 22217-5000		12. REPORT DATE May 1986
		13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT *(of this Report)* APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED		
17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*		
20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

TABLE OF CONTENTS

1.0 INTRODUCTION

Task allocation - The assignment of tasks to processors is an important problem in the design of distributed real-time systems. A task allocation scheme is required in order to produce a feasible partition of tasks across processors in the system, and to ensure high performance, especially for systems with real-time operational requirements. The task allocation problem for distributed systems has been studied by several researchers; [Chu80] contains a survey of various approaches.

One of the problems of current interest in real-time system design is the development of real-time Ada software for distributed systems. Several approaches have been proposed and are being studied, a survey can be found in [Arm84]. The approaches can be characterized as either

- *source code allocation*, the development of a multitasking Ada software which is then partitioned; this approach allows development and testing of the software as a whole before allocation;
- *target-code allocation*, where a compiler is responsible for performing task allocation, perhaps with some user-imposed contraints; or
- *separate program development*, where allocation decisions are made early in the development phase, and separate programs are developed.

The traditional approach of developing separate source programs for each processor in the distributed system requires the system designer to make early decisions on allocation, taking into account resource and performance constraints. This approach, however, increases the difficulty of software reallocation in the later phase of the software life cycle. Target code allocation schemes require a distributed target compiler which is used to generate separate object code files in allocating target code for each processor. There are two ways that the compiler can partition Ada application software: (1) being informed via pragmas about a predetermined partition scheme; or (2) analyzing the application software, and then applying a partitioning algorithm. The compiler required for this allocation scheme is complex and difficult to design, and presently not available.

The source code allocation approach has a number of important advantages. In an allocation scheme, it is preferable to place minimum design restrictions on software development, especially since the target architecture may, in most cases, not be known. It is also preferable to minimize the burden on the compiler. Additionally, since the underlying system constraints may vary, a good partition can be achieved only through iteration, and therefore, creating new partitions should be inexpensive.

One approach that meets the above objectives is source code allocation approach adopted by GTE Strategic Systems Division in its multicomputer software technology for Ada. This approach allows an application to be developed and tested as a single multitasking Ada program on the APSE (Ada Programming Support Environment), and then partitions and distributes the tested software to the distributed targets. Program partitioning is done at the source level, and the distributed software modules are compiled on the target machines.

GTE Laboratories is conducting research to develop a methodology for the partitioning of Ada source code to execute in a distributed environment. Two major tasks are involved in the development of such a methodology:

1. the formulation and selection of parameters that can be derived from the Ada source code, to be used in the partitioning process; and
2. the development of an efficient partitioning algorithm.

The approach taken closely follows that described in a previous report [GTE85]. The basic goal of the approach is to transform a given Ada program into a graph based representation, and then to apply a partitioning algorithm for task allocation. The graph based representation is similar to the specification schemes that are being investigated at the University of Texas at Austin [Mok84, Mok85].

In this paper, we describe the research efforts and progress towards achieving the above tasks. In section 2, we describe the partition problem and possibilities for meeting it. In section 3, we define partitionable units in Ada software. In section 4, we enumerate parameters derivable from Ada source code to be used in partitionings. Section 5 concludes the report with a description of planned future work.

2.0 PARTITIONING PROBLEM

The problem of allocating Ada source over distributed targets can be formulated as a graph partitioning problem. For purposes of task allocation, an Ada program can be represented by a graph $G = (V,E)$ as follows:

- the vertices of the graph G, i.e., the set V, represents the partitionable Ada program units; and
- the communication or dependency between units is represented by the edge set E.

Given this representation of the Ada program, weights are assigned to the vertices and edges; a weight $w(v)$ for vertex $v \in V$ represents execution characteristics of the Ada partitionable unit, obtained from such parameters as computation , memory and similar resource requirements. The weight $w(e_{ij})$ assigned to the edge between unit (vertex) v_i and unit (vertex) v_j represents the total communication cost between the two units; this weight is obtained from such parameters as the number of data elements transferred in a communication, and the number of messages required per transaction.

The partitioning problem for graph $G=(V,E)$ can be formulated as follows: determine a partition of V into m disjoint subsets $V_1, V_2, \ldots, V_m$ such that

$$\sum_{v \in V_i} w(v) \le K, \quad 1 \le i \le m, \text{ for some constant } K$$

and

$$\sum_{e \in E_{ij}} w(e) \le J, \text{ for some constant } J, \; E_{ij} \quad E \text{ and}$$

$$(v', v'') \in E_{ij} \Rightarrow v' \in V_i, \; v'' \in V_j \text{ and } V_i \ne V_j.$$

The partitioning problem, as formulated, aims at reducing the communication cost between the partitioned clusters, and also places a load balancing constraint on the clusters. These objectives are appropriate for the Ada partitioning problem as the performance of a system is affected by factors such as interprocessor communication delays, processor load, and the amount of parallelism that can be exploited.

The partitioning problem as formulated has been shown to be NP-complete [Gar79]; however, there are partitioning algorithms that use heuristics to obtain close to optimal results with acceptable algorithm performance [Pri84,Ker69].

The heuristic techniques reported in the literature can be classified into three categories [Lin81]: (1) constructive initial assignment, (2) iterative assignment-improvement, and (3) branch and bound technique.

The *constructive initial assignment* techniques are based on the concept of assigning one unit at a time to a particular processor until all the units are assigned. Algorithms vary on the order in which the units are assigned and the criteria used to select the processor.

The *iterative assignment-improvement* techniques start with an initial assignment and the next assignment is generated by making a small improvement to the initial one [Ker69]. The algorithms terminate when no improvement can be discovered or after a predetermined number of iterations.

The *branch and bound* methods are based on the concept of doing an implicit search of a decision tree. Algorithms use different heuristic methods for deciding which branch in the decision tree to follow and for pruning possible solutions.

The iterative assignment-improvement algorithms are used more than the other two techniques. In general, the branch and bound algorithms are too slow for large applications and the constructive initial assignment algorithms do not generate partitions as good as the other two techniques.

One of these types of algorithms will be customized and applied for the Ada source code partitioning problem. It is expected that the number of vertices in a graph obtained from Ada source will be large, and therefore heuristics will need to be developed not only for creating good partitions, but also for partitioning in an acceptable amount of time.

3.0 PARTITIONABLE UNITS

In our framework of source code partitioning, we discuss what constitutes a partitionable unit of an Ada program. Since the partitioned software has to be compiled on each processor, the partitioned units must be separately compilable. In Ada, there are four kinds of program units that can be separately compiled. They are tasks, subprograms, packages, and generic units.

Subprograms are the basic executable units of Ada programs. They can be procedures or functions. A subprogram communicates with outside entities via global declarations or parameter passing upon its invocation and termination.

In Ada, a collection of logically related entities can be encapsulated in a package. A package allows its entities to communicate with an entity outside the package via global declaration or by the import and export mechanism. The entities declared in the visible part of the package specification may be used outside the package. And entities in another package may be used by establishing the visibility through the with clause.

Unlike subprograms and packages, tasks operate in parallel with other program units. The main program unit is implicitly considered to be a task. In Ada, task interaction is handled by treating each task as a communicating sequential process [Hoa78]. The tasks are synchronized in time when they communicate. The explicit synchronization is known as a rendezvous. Similar to package specification, a task specification defines the communication entries available to other tasks.

These three kinds of program units can be introduced in the declarative part of any unit. This makes the communication among units non-trivial. We will discuss this in a later section.

Some distributed Ada systems allow partitioning on task boundaries only. That kind of approach appears to have achieved a synergy between Ada's units of concurrency and the underlying system's unit of concurrency, the processor. However, this approach requires all code being partitioned be encapsulated by a task. This requires early partition to make sure that tasks are designed at the appropriate place. This does not meet our objective of making minimum design restrictions. In some applications, limiting interprocessor interface to only task rendezvous may be unnatural; the interprocessor interface may be better represented as a call to a procedure inside a package and not a call to an entry for a task.

We propose that partitioning be allowed on these three kinds of program units boundaries. We do not explicitly include generic units as partitionable units. We can view generic packages/subprograms and their instantiations as packages/subprograms in the partitioning scheme. Since the partitioned units have to be compiled on the target machines, we may require the partitionable units to be designed as Ada compilation units for distribution purposes. This does not impose any syntactic restriction or any design restrictions since Ada program units can be submitted as separate compilation units or as one compilation. However, it must be understood that a compilation unit does not have to be a partitionable unit. In Ada, each compilation unit specifies the separate compilation of a construct that can be a subprogram declaration or body, a package declaration or body, a generic declaration or body, or a generic instantiation. A compilation unit can also be the body of a task unit.

4.0 PARTITION MODEL

We have discussed the general partition problem and our proposed partitionable units in Ada in previous sections. We now propose a model for partitioning an Ada program for distributed targets.

The first step in our modeling is to represent the interunit communication as a graph G = (V,A), where V is the set of vertices representing the partitionable units of an application, and A is the set of arcs representing the communication. Our next steps will be examining how to assign weights to the vertices and the arcs.

4.1 INTERUNIT COMMUNICATION

A unit can be a subprogram, a task, or a package of data objects. There are four kinds of communication among these units.

The first kind is *subprogram invocation*. A subprogram's execution is invoked by a subprogram call from another subprogram or a task. After the association between formal parameters and actual parameters is established, the execution control is passed to the called subprogram. Upon completion, control is returned to the caller. The subprogram invocation follows a single thread of control. The communication cost is incurred at invocation and completion.

The second kind of interunit communication is *task rendezvous*. Different tasks execute independently, except when they communicate. A task entry can be called by another task. The communication is established when the called task accepts the call. If the entry has parameters, values are communicated between the tasks. After this synchronization, the task issuing the entry call and the task accepting the call continue their execution independently.

Task activation/termination is another kind of communication related to tasks. This is an implicit communication in the control flow of the task dynamics. The initial part of task execution is called activation. A task is activated as a result of the elaboration (execution) of the declarative part of its parent task or as a result of the allocation of a new task. A task is said to be terminated when it is completed (it finishes its last executable statement) and all its dependents are terminated. Therefore, upon termination, a dependent task needs to communicate its state to its parent. This kind of activation/termination communication occurs between a task and any other kind of partitionable unit.

Data reference/modification is a different kind of interunit communication that is not explicit. A partitionable unit can refer any visible data defined in other units. A data definition in a unit is made visible to another unit either by scope rules or with clauses. Data reference/modification is purely data flow; there is no control flow involved.

Two units with any of these four kinds of communication will experience some network delay when they are allocated to different processors. Over the same network, different kinds of communication take different amounts of time. We will discuss the weights on these kinds of communication in the partitioning scheme.

Although communication is bidirectional, we like to assign direction to it for analysis purposes. We say a communication is from unit A to unit B if unit A initiates the communication.

We assign a weight to every communication from unit A to unit B if they are assigned to different processors. The weight of an interprocessor communication depends on the number of messages required for such a communication. In the case of subprogram invocation and task rendezvous, the number of arguments in the call has to be taken into account for the weight of the communication.

For a call to a subprogram on a different processor, two messages are needed: a "call" message containing the IN parameters, if there are any, and a "return" message containing any OUT parameters.

In addition to the "call" and "return" messages, there are two more messages needed for each normal task rendezvous [Wea84]. After a "call" message is sent to the accepting task and when it is ready to accept, an "accept" message is sent. If the calling task still desires rendezvous, a "confirm" message is sent to the accepting task. Figure 1 depicts the message passing required for task rendezvous. The "accept" and "confirm" messages are less complex than the "call" and "return" messages that contain IN and OUT parameters.

In the event of elaboration, a task sends an "elaborate" message to all its dependent tasks and waits for an "active" message from each of its dependents. When a task completes its last executable statement, it waits for a "terminate" message from each of its dependents before it

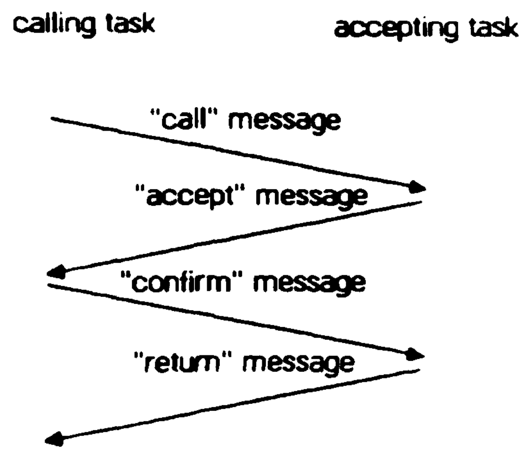


Figure 1. Messages Required'for Task Rendezvous

terminates. Therefore, there are totally three messages between a parent and each of its dependents required for activation and termination.

For data reference/modification between two units on different processors, a "request" message is sent from the initiator and a "response" message is returned.

The weight of an edge from unit A to unit B is the sum of the weights of all communication from A to B.

4.2 COMPUTATIONAL COMPLEXITY OF THE UNITS

Several program complexity metrics have been developed for various purposes, such as maintainability, and understandability. For partitioning purposes, we are interested in the computational complexity of a unit. The complexity measure includes the unit's time and space requirements. Knowing the compelxity of each unit, we may be able to achieve better load balancing for the processors in the system. We use a simple definition of load balancing. Load balancing is an assignment of units to processors such that the time and space requirements are evenly distributed to each processor in the system.

A unit, except a package of data, contains a code portion and a data portion. A suitable metric for measuring the space requirements of the code portion of a unit might be the number of machine instructions. However, the number of machine instructions generated from Ada source is compiler dependent. In general, the expansion ratio of the number of machine instructions per line achieved by a compiler is not known. GTE-GS SSD has done some work in measuring emperically the expansion ratio of a group of compilers [Che86]. Since we are using a less strict definition of load balancing, that is, we are not aiming at an optimal assignment, the number of source lines could be a good estimate of the space requirement for a unit's code portion. Similar to code space requirements, a unit's data storage requirement is compiler dependent. At this point, we are going to use a set of assumptions about Ada data type storage requirements.

The time complexity of a task or subprogram unit that does not contain a loop statement can be measured by the number of machine instructions generated for the unit. To analyze a unit with loop statements is nontrivial. A loop statement without an iteration scheme can be repeatedly executed until a transfer of control occurs. In most cases, how many times a loop is going to be executed cannot be predicted until the transfer of control occurs. Even for a loop statement with a "while" iteration scheme, it is difficult to estimate the number of iterations. We can only be certain of the number of iterations in a loop statement with a "for" iteration scheme. We can view all loop statements with or without iteration schemes as a sequence of code to be executed periodically. Therefore, the time requirement of a unit can be estimated by the number of source lines without regard to loop statements.

5.0 CONCLUSION AND FUTURE PLANS

In this paper, we have discussed the problem of partitioning real-time Ada software for distributed targets and adopted source code allocation as an approach to the problem. In this approach, code for an application is developed and tested as a single Ada program, and then partitioned and distributed to distributed targets, where compilation takes place at each location. A partitioning methodology for Ada programs has been outlined.

The next steps in the research program will concentrate on testing the ideas outlined in this paper. Specifically, the following tasks will be pursued:

- Guidelines will be developed for use in translating an Ada source program into a graph for partitioning; this will determine the validity, as well as the ease of derivation of the parameters discussed in this paper.
- An efficient graph partitioning algorithm will be developed. This will entail the derivation of a set of heuristics to enable the generation of "good" or "close-to-optimal" partitions without prohibitive computation costs.
- Finally, the performance of the partitioning methodology developed will be assessed to determine its effectiveness in generating high performance partitions.

REFERENCES

[Arm84] Armitage, J. W. and J. V. Chelini, "Ada Software on Distributed Targets: A Survey of Approaches," GTE Government Systems, Strategic Systems Division TN 84 807.6, October 1984.

[Che86] Chelini, J. V., E. B. Hudson and S. M. Reidy, "A Preliminary Study of Ada Expansion Ratios," *ACM Software Engineering Notes*, Vol. 11, No. 1, January 1986, pp. 35-46.

[Chu80] Chu, W. W. et al., "Task Allocation in Distributed Data Procession," *IEEE Computer*, Vol. 13, No. 11 (1980), pp. 57-69.

[Gar79] Garey, M. R. and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[GTE85] "Task Allocation in Distributed *Hard* Real-Time Systems," *GTE Laboratories Technical Proposal No. RL 5029* (1985).

[Hoa78] Hoare, C. A. R., "Communicating Sequence Processes," *Communications of ACM*, Vol. 21, No. 8, August 1978.

[Ker69] Kernighan B. W. and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, February 1970, pp. 291-307.

[Lin81] Lint, B. and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithm," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 2, pp. 174-188.

[Mok84] Mok, A., "The Decomposition of Real-Time System Requirements into Process Models," *Proceedings of the 1984 Real-Time Systems Symposium*. pp. 125-134.

[Mok85] Mok, A., and S. Sutanthavibul, "Modelling and Scheduling of Dataflow Real-Time Systems," *Proceedings of the 1985 Real-Time Systems Symposium*, pp. 178-187.

[Pri84] Price, C. C., and S. Krishnaprasad, "Software Allocation Models for Distributed Computing Systems," *Proceedings of the 1984 Distributed Computing Systems Conference*, pp. 40-18.

[Wea81] Weatherly, R. M., "A Message-based Kernel to Support Ada Tasking," Gensoft Corporation, Pittsburgh, PA, 1981.