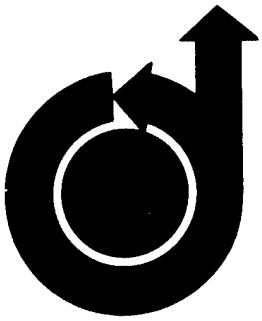
4



TECHNICAL INFORMATION SERVICE

19980513 045

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC QUALITY INSPECTED 4

PLEASE RETURN TO:

BMD TECHNICAL INFORMATION CENTER
BALLISTIC MISSILE DEFENSE ORGANIZATION
7100 DEFENSE PENTAGON
WASHINGTON D.C. 20301-7100

U02201

AMERICAN INSTITUTE OF AERONAUTICS AND ASTRONAUTICS
555 WEST 57th STREET NEW YORK, N.Y. 10019 212/247-6500

Accession Number: 2201

Title: Assessment of the Development of a Tracking System Using Concurrent Ada (paper)

Personal Author: Lemanski, W.J.; Hartrum, T.C.

Corporate Author Or Publisher: AIAA

Descriptors, Keywords: Methodology Development Communication Tracking Algorithm Cyclic Integration Parameter Allocation Ada Software Assessment Kalman Filter Code

Pages: 008

Cataloged Date: Jun 18, 1990

Document Type: HC

Number of Copies In Library: 000001

Original Source Number: A90-30722

Record ID: 21120

Source of Document: AIAA

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

AN ASSESSMENT OF THE DEVELOPMENT OF A TRACKING SYSTEM USING CONCURRENT ADA

Walter J. Lemanski and Thomas C. Hartrum
Department of Electrical and Computer Engineering
School of Engineering
Air Force Institute of Technology
Wright-Patterson AFB, Dayton, Ohio, 45433 [1]

A90-30722

1989

Abstract

Parallel processing holds great promise for meeting the demands of complex real-time systems such a those required for the Strategic Defense Initiative. Full realization of the capabilities of parallel computing will require the use of design methodologies and programming languages specifically designed for that purpose. This paper describes the development of a representative real-time system from problem analysis to final implementation on a parallel machine. The problem selected for implementation is a multiple model Kalman filter tracking algorithm. The system was designed using the Layered Virtual Machine/Object Oriented Design methodology proposed by Nielsen and Shumate [11], and was implemented in Ada on a 16 processor Encore Multimax. The paper discusses speed-up results and includes commentary on the methodology used, the advantages of using Ada in terms of programmer productivity, portability, and reusability of the code.

1 Introduction

Although successful realization of the goals of the Strategic Defense Initiative will require significant advances in many areas of science and engineering, it is now generally accepted that computers and their associated software have become the "long pole in the tent." SDI software will be the most complex ever developed. The operational system will have to run in real-time and be extremely reliable. In addition, extensive ground-based simulators will be necessary.

Parallel processing shows great promise not only for meeting the computationally intensive real-time deadlines of SDI software, but also for substituting processing power for software complexity by allowing the use of simple compute-intensive algorithms rather than complex optimized code. The use of parallel processors can also increase the reliability of the system by distributing the computing load over many processing units. Finally, parallel systems provide the best alternative for the massive amounts of computing power necessary to execute the ground-based simulations.

In order to exploit the power of parallel processing effectively, it is necessary to have a software programming language which efficiently abstracts parallel algorithms. The Ada language was specifically designed for use in parallel real-time software development. It explicitly supports independent communicating processes through the use of the task and machine-level operations with representation specifications. In addition, Ada is a third generation high-order language with extensive support for software engineering concepts.

The purpose of this research was to explore the possibilities presented by parallel processing and the Ada programming language and to determine how they could benefit the development of SDI software systems. To do this, a non-trivial application was selected to be designed and implemented as a parallel Ada system. The Air Force is developing a tracking system which uses a forward looking infrared sensor together with a tracking algorithm to provide target information to a laser pointing system. The purpose of the tracking algorithm is to detect vehicle movement in order to keep the target centered in the tracker's field of view, simultaneously pointing the laser.

A promising tracking algorithm is being developed at the Air Force Institute of Technology (AFIT) based on Kalman filtering techniques [8,9,10,14]. The heart of this tracking system is a Kalman filter algorithm which processes inputs from measurement devices together with the knowledge of applicable device dynamics, statistical descriptions of noises, measurement errors, modeling uncertainties, and initial conditions, to produce an optimal estimate the current and predicted future position of the target. The tracker is made even more accurate through the use of a multiple model adaptive filter system (MMAF). The MMAF is composed of a group of Kalman filters constructed with varying target characteristic specifications. The outputs from the filters are arbitrated by giving varying weight to each filter, based upon its predictive ac-

[1]This work sponsored by the Strategic Defense Initiative Organization

U.S. Government work not protected by U.S. copyright.

curacy. The result is an estimate more accurate than a single filter could provide. However, this increased accuracy comes at a cost of increased computational loading. The size and complexity of the algorithm makes it a good test case for parallelization. In addition, it is an excellent example of the type of ground-based simulation SDI will require.

The system was implemented on an Encore Multimax 320, a tightly-coupled multiprocessor architecture based on the National Semiconductor 32332 processor. The Multimax 320 at AFIT is configured with 16 processors and 32 Mbytes of main memory. The system operates under a single copy of UMAX (which is a derivative of UNIX) which is accessible by all of the processors simultaneously. UMAX implements the concept of multi-threading, allowing it to support multiple, simultaneous streams of control. Interprocess communication occurs through shared memory The operating system also supports process migration, making dynamic load balancing possible [4].

The Encore Ada run-time environment makes true concurrent Ada possible. Rather than using some single-processor interleaving scheme, Encore Concurrent Ada allows the user to specify the number of independent processes desired. Tasks are then scheduled on a first-come first-served basis from a single queue when a process becomes available. Tasks priorities are supported through the use of multiple task queues, one for each priority level. The user has no explicit control over task allocation or scheduling. The processes are assigned to processors by the operating system. Tasks are assigned to available processes by the Ada run-time system [5].

The ability to do true concurrent programming in the Ada language is relatively new. Many software engineering issues in this area still need to be resolved. The purpose of this paper is to discuss how we resolved these issues in the context of a functional concurrent Ada implementation. The issues discussed include: language partitioning, multitasking design requirements, problem decomposition, and a parallel Ada design method. A description of the implementation and testing methods used is also included. The paper concludes with a discussion of project results including an analysis of both the design methodology and the adequacy of the Ada language for parallel software development.

2 System Analysis and Design

2.1 Language Partitioning Strategy

Despite the fact that the task construct was specifically included in the language to support parallel processing, there is some disagreement as to the exact time and method that should be used for partitioning Ada programs. There are two times in the development process when software can be partitioned. The two corresponding strategies are known as pre-partitioning and post-partitioning [21]. In post-partitioning, after the program is designed and written as if for a single processor, the desired partitioning is specified using separate software tools, or is accomplished automatically by the distributed operating system. Pre-partitioning begins at the very start of the design process. A particular construct of the programming language is selected as the basis of parallelization and used to encapsulate each of the system parts that will run in parallel.

One advantage claimed for the post-partitioning strategy is that it promotes portability by allowing the same program to be mapped onto different hardware configurations. Another is that, because the program is written to run sequentially, there are no restrictions on how the language is used. Finally, it avoids concern that the Ada language does not contain facilities for specifying the configuration of the software over the underlying hardware.

Automated post-partitioning does not currently exist, nor is one likely to be developed in the near future. Manual post-partitioning is possible using tools such as Honeywell's Ada Program Partitioning Language [2,3]. This language allows the mapping details to be expressed completely separate from the program. However, the efficiency of the tool remains to be demonstrated.

Correct use of the Ada language in the pre-partitioning strategy can provide the advantages claimed for post-partitioning. Operations that lead to hardware dependency in a parallel system, such as task synchronization and dispatching, storage management, and exception handling, are handled in Ada by a machine-specific run-time environment and not compiler-generated code As a result, concurrent software written in Ada can be made portable if properly implemented.

Concern about restrictions on language use in this strategy are also unfounded. Although Ada code must be encapsulated in the task construct to run concurrently, the uniformity of the language reduces the impact on the programmer's productivity. For example, task entry calls mimic procedure calls and can be used in the same manner procedure calls would be used in a sequential program. Within the task, all operations are available that would be available in a corresponding sequential program. Therefore, the use of the task construct places no unreasonable restrictions on the programmer.

The lack of configuration facilities in Ada can be overcome through the use of data driven design [13]. Rather than depending on manual partitioning, which would require configuration facilities to schedule and control independent tasks, data driven design depends on the run-time environment scheduler to control access to processors, based on which tasks are ready to run and their relative priorities. Because the Ada rendezvous mechanism blocks tasks that are awaiting data, the flow of data through the system determines which tasks are ready to run. By controlling this flow of data, the designer can control the

scheduling of tasks without extra-language configuration management facilities.

Data driven design results in minimum processor idle time because ready tasks do not have to wait for the start of their frame time as in the case of the cyclic executive. Instead, they are scheduled as soon as a processor is available. In addition, data driven designs adapt automatically to changes in hardware resources, timing, or processing requirements because scheduling decisions are made in real-time by the run-time system based on the availability of data and processing resources. This method also eliminates the need for time frames and, therefore, the possibility of frame overrun. On the other hand, timing problems are still possible if processing requirements outpace available processing resources. Because high priority tasks are always scheduled first, any timing problems will show up in low priority tasks first. The result is a throughput problem in the overall system. Since the processor idle time is already at a minimum, the only way to solve this problem is by increasing processor resources or decreasing computational demand.

Pre-partitioning is the superior strategy for parallel design in Ada because it promotes early examination of critical design issues and eliminates the overhead of separate partitioning specifications. In addition careful design can yield the benefits of the post-partitioning strategy including: portable software, freedom of language use, and adequate concurrent process control using the concept of concurrent design.

Given the pre-partitioning strategy and the Ada language, partitioning can occur at four language levels [2,3]. Because two of these, partitioning on any source program construct and extending Ada, require facilities outside of those provided by the Ada language, these two methods will not be considered here. Of the remaining two methods, the first consists of writing a separate program for each processor. This method was commonly used in the past because of a lack of language constructs designed specifically for parallel processing. It is inefficient and has several disadvantages. First, this approach ties the software closely to the underlying hardware. If the hardware is later changed, the software must often be redesigned and rewritten. Any attempt at reallocating functions among processors will also require redesign.

Another problem is that much of the reliability gained from the use of high order languages is lost. In Ada, semantic rules are enforced by the compiler only within a program, not across program boundaries. Compilers may even generate different internal structures for logically identical data types in each of the programs leading to hidden problems when data is exchanged between them. Finally, communications capabilities in the Ada language are strong for passing data within programs, but the only facility available for communication between programs is inefficient and difficult I/O transfers.

The other possible alternative is to partition on task boundaries by encapsulating the code representing each problem fragment within a task. The entire system is contained within a single program with the task acting as the basis for concurrency. This method increases software portability, and reallocation of functions within the program can be handled by the run-time environment or with only localized changes to the affected task. The fact that the whole system is contained within a program allows for the full range of semantic checking and makes Ada's extensive communication capabilities available to the programmer. These advantages make the use of the task construct the superior option.

2.2 Multitasking Design Issues

The use of the task construct as the basis for concurrent processing introduces some new design issues. It is generally accepted that strong cohesion and loose coupling leads to more structured design, easier modification, and higher maintainability. The idea of cohesion can be applied to tasks, but the coupling concept must account for the interdependencies that result from concurrent operations. In this context, coupling can be considered at two levels. In the first case, between subprograms and tasks, coupling can be evaluated in the standard way.

The differences in the concept of coupling during concurrent operations manifest themselves in the second case, task to task interaction. This is called concurrency coupling. Tasks are considered tightly coupled if one calls the other's *entry* directly. There are various degrees of tightly coupled tasks. The tightest occurs during a rendezvous where *in out* or both *in* and *out* parameters are included in the call. In this case, the calling task requires a reply and the two tasks must remain synchronized for the entire period of data transformation. A lessor amount of coupling occurs when only *in* or *out* parameters are in the call. Here, the two tasks are only synchronized long enough for data to be copied from one to the other. A parameterless call represents the least amount of coupling. Tight coupling should be avoided whenever possible because periods of synchronization eliminate independent operation and thus reduce the efficiency of parallel processing. Loose coupling is achieved through the use of intermediary tasks between the caller and called task pair. These tasks perform a buffering function to ensure that the two main tasks can continue processing unhindered by unnecessary synchronization time.

Three varieties of intermediary tasks are identified in [11]. A *buffer* task is a server only. It contains an entry to accept data from a producer and an entry to provide data to a consumer upon request. In between producer and consumer calls, the data is stored internal to the task. A *transporter* is strictly an active task. It requests data

via an *entry* call to the producer task and then outputs the data via an *entry* call to a consumer. A *relay* task is a combination of the two. It waits until called by the producer and then immediately calls the consumer. Intermediary tasks are used in various combinations to achieve the desired degree of coupling between producer and consumer tasks. They are also useful to control caller/called relationships, and should be tailored directly to specific design requirements. A good concurrent design should have a balanced use of intermediary tasks with no cyclic dependencies and a minimum amount of busy waiting. It should also minimize the amount of processing done during a rendezvous (statements within *accept* blocks) and ensure appropriate modes are used for *entry* parameters.

The introduction of concurrency into the system and the need to reduce task coupling must be balanced against the overhead of the resulting tasks. This includes task activation, termination, scheduling, dispatching, allocation of task control blocks, and, when necessary, context switching, exception handling, management of entry queues, and rendezvous. The amount of overhead associated with each of these operations is dependent on the specific run-time system implementation, but it must always be considered.

2.3 Problem Decomposition

The first step in any parallel software project is to determine how the problem can be decomposed into independent processes. This section describes three methods of problem decomposition, using the terminology developed in [12]. The three methods are: relaxation, pipelining, and partitioning.

Relaxation consists of dividing processing into independent functions, each of which operates on the same input data and performs a complete function. They are not dependent on data from each other, and no synchronization is required between them. This type of decomposition is ideal when multiple tasks are being performed at the same time, a common occurrence in real-time systems.

Pipelining consists of dividing the problem into functions that follow each other sequentially, each performing its portion of the computation on data from the one before it. Each segment must produce results at the same rate or the system will bottleneck at the slowest process. This works well for problems where complex operations follow each other sequentially over repeating data inputs.

Partitioning is the third method of decomposition. Instead of each processor performing different computations, groups of processors work simultaneously on subparts of the same problem. This method differs from the other two in that partitioning is done by dividing the problem data domain rather than its algorithmic functions. The result of the divided data domain is called the grain, and grain size has an important effect on the efficiency of the resulting code. Partitioning is most effective for homogeneous operations performed on large data sets, typically identified through iterative structures. The goal is to find the particular loop which constitutes the greatest computational loading and divide it across the available processors.

There are three major limiting factors that impact the decomposition of a problem and the efficiency of parallel software. The first is the computational overhead added by parallel software. This overhead comes in one of three forms: tasking, communication, and synchronization.

Tasking overhead consists of the work necessary to perform the tasking functions previously described. It is a function of the number of independent processes in the system and the efficiency of the run-time environment. Communication overhead is a function of the amount of data passed between processes and the communications efficiency. It includes the overhead inherent in the programming language (*e.g.*, in Ada, subprogram invocation, task rendezvous, task activation and termination, and data reference and modification), and the computational cost and time delay incurred by physical message passing. Synchronization overhead is a function of the number of times individual processes must suspend operations to communicate with other processes and the amount of time in that suspended state. The efficiency of the load balancing has the greatest effect on this type of overhead.

The second limiting factor in parallel speedup is the percentage of sequential operations in the problem that cannot be parallelized in any manner. These operations must be performed on a single processor while others are held idle, reducing the efficiency of the overall algorithm.

The final limiting factor is data contention, when several processors require access to the same global data element at the same time. On a shared memory machine, processes queue up to obtain the data, resulting in a serialization of the parallel tasks. On a loosely coupled system, data contention results in increased message passing and increases the possibility of multiple copies of the same global variable existing in different states. This effect is called software lockout and must be minimized whenever possible.

The Kalman filter tracking algorithm was decomposed on two levels. Relaxation was used at the highest level to separate the individual filters in the Kalman filter bank and to separate the functions of the simulator. At a lower level, partitioning was used for some of the complex matrix operations required by the algorithm including matrix multiplication and Cholesky decomposition. These operations were often performed on large data sets, justifying the added overhead of parallelization.

2.4 Software Design Methodology

Parallel programming and the Ada language place certain demands on a software design method. The method must include a means of describing concurrent processes and

the communication between them. In addition, it must support the software engineering features of Ada including packages, generics, and advanced data structures. Finally, it should support software pre-partitioning using the task construct. One method specifically developed for designing large, real-time distributed systems in Ada is the Layered Virtual Machine/Object-Oriented Design (LVM/OOD) method [12]. It provides capabilities for describing concurrent processes and the communication between them and also supports real-time considerations. Its basis for concurrent processes is the Ada task, groups of which are encapsulated into packages.

Good software design requires the successful integration of algorithms and data structures. Both of these components are of equal importance, but they often raise differing concerns. LVM/OOD deals with these concerns by combining two design concepts: layered virtual machines and object oriented design (OOD).

Layered virtual machines are abstractions of algorithms into independent processes capable of operating in parallel. These processes are virtual because they are not associated with underlying hardware. The run-time environment is responsible for the binding of processes to processors. The use of layering creates a hierarchical set of modules that defer implementation details and support information hiding.

OOD is used to abstract problem space objects into software data structures and their associated operations. Object abstractions can be either object managers which encapsulate data structures or type managers which encapsulate definitions that form templates for the creation of data structures. These managers provide the means for describing data and are especially important in data driven designs.

The layered virtual machine and object oriented design concepts are combined into an eight step design methodology. The first step is to determine the hardware interfaces to the control system, illustrated with a context diagram. Each hardware device is depicted separately and the internal control system is represented as a single entity. High-level inputs and outputs are labeled on the interfaces.

In the second step, each of the external devices, or edge functions, is assigned a separate process, a simple device driver with the bare minimum of instructions. Third, the internal controller is decomposed into its primary components using data flow diagrams (DFDs). Complex components can be further decomposed using hierarchical levels of DFDs.

The fourth step is to determine concurrency among the controller components, to abstract these components into processes that will run independently. Concurrency considerations will often lead to grouping components into a single process. Nielsen and Shumate provide several rules to assist in this. Functional cohesion suggests that closely related functions can be grouped into a single process to reduce overhead. Temporal cohesion collects operations that occur during the same time period or after the same event. In both cases, significant overhead can often be saved without appreciable loss of performance. However, some functions should be left as separate processes. Time-critical components should be implemented as separate processes. Periodic functions should not be combined with operations that run in differing periods. Finally, background processes should also be separate to preclude interference with time-sensitive ones and to provide for the best use of excess processor time.

This fourth step focuses only on the concurrency of the high-level components identified in the system DFDs. Nielsen and Shumate recommend against having more than one level of concurrent tasks, but ignoring lower level partitioning may forgo performance gains. This research examined further decomposition, resulting in several matrix operation routines that increased overall system parallelization.

The fifth step is to determine what type of interfaces exist between the processes. These interfaces define the communication between the processes and can be one of several forms: messages where a reply from the receiver is required, data transfers where no reply is required, signals used to coordinate on the occurrence of certain events, and shared data access. The type of communication determines the amount of coupling between processes. Messages that require replies have the highest coupling followed by simple data transfers. The least coupling is caused by event signals. Shared data access requires additional attention and must be protected by some intermediate task to provide for mutual exclusion. Once these interfaces are identified, processes and their corresponding interfaces are depicted using a process structure chart.

Step six seeks to reduce coupling by introducing intermediary processes into the design. First, the processes are translated into Ada tasks. Between tightly coupled tasks, additional tasks are added whose sole purpose is to facilitate communication and thereby decrease coupling. These intermediate tasks can be any combination of the three types described earlier. The result of this step is the Ada task graph.

Step seven is to encapsulate the tasks into packages. At the same time, the objects used for communication between the tasks should be abstracted into data objects and encapsulated into packages as well. Nielsen and Shumate suggest that these be placed in packages to increase modularity, portability, and reusability. They also provide several rules for task encapsulation. Tasks should be grouped into the same package if they have similar functions or if their general nature makes them good candidates for reuse. Coupling between packages should be minimized with respect to data types, operators, and constants by localizing these items to the package or group of packages in which they are actually used. Finally, the package should mini-

mize the visibility of task entries to that which is essential for package interfacing. The products of this step are an Ada package graph and the corresponding OOD diagrams

The final step is the further decomposition of large tasks. This may result in another level of concurrency which can be depicted using process structure charts and Ada task graphs, or it may result in a sequential decomposition which is described with structure charts. The results of this step are shown using OOD diagrams.

The complete methodology was used to design the Kalman filter tracking system. Interested readers are referred to [7] for the results.

3 System Implementation and Testing

This section describes the process used to implement the Kalman filter tracking system from the design developed using LVM/OOD. A top-down approach was used to build a skeleton of the system to outline and test system interfaces. These interfaces were used as a basis for a bottom-up construction of the final system. Testing was completely integrated with the implementation, occurring at the end of each phase of the development.

The first phase of the top-down approach began with the coding of the system packages. Global type descriptions were declared in the package specification of the package in which they were first referenced, and simulation model constants were declared in a master package. With these high level data descriptions in place, a skeleton of the complete system was developed using the top-level tasks. Each task declaration was complete with all of its entries and their corresponding parameters. The task bodies were coded as shells containing only the *entry* calls and *accept* blocks from the final implementation. An output statement was included before and after each *entry* call and *accept* block describing the current state of the task. All variables local to the task bodies were declared and those used for communication parameters were initialized to zero.

Once the coding was complete, the skeleton was compiled using the already established order of compilation based on the package dependencies identified in the design. This provided a first check of package dependence, Checked the actual parameters in the *entry* calls against the formal parameters in the *accept* statements, and indicated any improper parameter modes.

Once compiled, the skeleton system was executed on a single processor. This caused the elaboration of all of the task and object declarations and activation of all the tasks. It also checked the system for possible deadlock conditions. Although no data was processed or passed, all rendezvous took place and were documented by the output of the task state descriptions. The resulting record of system operation provided clues as to the state of the system at the point at which deadlock occurred. This information was useful in reordering *entry* calls to eliminate the deadlock problem.

Despite the usefulness of the system state output technique, it does have limitations. There is no guarantee of what an output device driver will do when a concurrent system deadlocks. Messages may be lost and the resulting state record may not be complete. Even a deadlock-free system at this point does not guarantee continued correct operation once the skeleton has been filled in with all the required processing. Because of the non-deterministic nature of the tasking model, additional processing load may lead to a different task scheduling order which may present opportunities for deadlock that did not exist in the skeleton. Only careful analysis of the system design and the results of tests can lead to a dead-lock free system. Nonetheless, until better tools for following tasking flow of control are made available, the system state description log is one of the best methods of finding concurrent software communication errors.

The second stage of the top-down part of the development involved adding the subprogram specifications to their package specifications and subprogram stubs to the corresponding package bodies, and recompiling. Package dependencies were checked again, and the formal and actual parameters in the subprogram specifications and calls were compared for type mismatches. The skeleton system was executed again to force elaboration of the subprogram local variables.

The completed skeleton provided a framework for further development by documenting the interfaces between the high-level processes. This knowledge formed a basis for further development of the system from the bottom up. The first step was the coding of several reusable matrix operation routines. Because these operated on large matrices, they were parallelized to provide better response times. A single, shared copy of the data was used to eliminate rendezvous overhead, and the number of tasks spawned to complete the operation was varied dynamically,1 based on the size of the particular matrix.

Using the matrix routines as atomic operations, the rest of the system was developed from the lowest level subroutines upward. Difficulties in debugging were caused by the fact that the Encore run-time system would not propagate unhandled exceptions out of tasks. When an error occurred within a task without an exception handler, the run- time system would simply hang without an error message. Often this required moving the code to another development environment in order to determine the error. The portability of Ada was a real advantage here, as the entire Kalman filter tracking system could be ported with minimal changes to the code.

4 Conclusions

One objective of this research was to examine the software engineering issues surrounding the use of Ada for concurrent software systems. The implementation of the Kalman filter tracking system proved that the task is a viable construct for parallel process partitioning. In addition, it proved that careful problem decomposition will result in more than adequate computational load to outweigh the inherent overhead of the tasking model and the Ada run-time system. The use of intermediary processes proved very useful in reducing synchronization overhead and, thereby, increasing the parallel efficiency of the system.

Nielsen and Shumate's [12] design methodology proved to be an effective means of documenting a parallel Ada design. It provides good rule-based decision making support at both the system and detailed design levels. The graphical tools are adequate to describe the state of the system at each stage of development, and each level of problem abstraction is supported. The method does tend to be more functionally oriented in some stages (neglecting the OOD portion of the methodology), and it was supplemented in those areas by material from [1].

Two major areas of difficulty were encountered in the design phase of the project. The first was the discovery in the implementation phase that the one part of the algorithm was a significant roadblock to parallel speed-up with the tracking system. Although some problem was expected based on the results of the initial decomposition, the magnitude of the bottleneck was unexpected. Clearly, it would have been better to recognize the severity of the problem earlier in the design process or even in the initial analysis. While the factors limiting decomposition efficiency were discussed early in the guidelines, no real method was available for discovering the relative magnitude of these factors.

In a parallel environment, such a method cannot be based solely on standard algorithmic complexity analysis. It must include knowledge of actual module execution speeds and actual time delays for task allocation, scheduling, and communication. Whether such information could be determined during analysis and design, without some degree of code test cases, is uncertain. Also, the availability of such information presupposes knowledge of the specific implementation hardware which may not be available in the early stages of the design. The form of such a complexity analysis method, as well as where in the development cycle it should fit, are areas which require further study.

A second limitation to the guidelines was the lack of a method for specifying independent process run-time interaction graphically. This would have been very helpful in identifying possible deadlock situations and would have provided a means of showing data dependency among tasks which would have helped in analyzing task coupling. A very complex problem, involving many layers of tasks, would be impossible to comprehend without some graphical display of task interaction. An automated graphics tool would be very helpful in this area.

The second objective of this effort was to determine the adequacy of the Ada language for parallel software development through the implementation of a real-world problem. The implementation of the Kalman filter tracking system highlighted a very important distinction between the adequacy of the language itself (as described by MIL-STD 1815A) and the adequacy of the tools (compilers, debuggers, run-time systems, etc) currently available to support it.

The Ada language proved to be an excellent means of abstracting a parallel problem. The task construct is ideal for encapsulating independent processes, while the rendezvous provides the means for both task synchronization and communication without resorting to machine dependent parallel language constructs. The ability to dynamically spawn tasks was useful as a load balancing tool. It made it possible for a generalized routine to vary the number of tasks spawned based on the size of the particular data structure being operated on.

Aside from the task construct, other features of the language made implementation more productive as well. The use of the package construct greatly assisted in modular development. The use of generics and unconstrained arrays made it possible to construct general support routines capable of operating on a wide range of data forms. Finally, the standardization of the language was itself an advantage. Because of the difficulties encountered with the debugging tools, four different Ada environments resident on four different hardware architectures were used during the development process. The use of Ada made it possible to move modules of code freely between these environments on a frequent basis with a bare minimum of changes.

While the Ada language provided ideal support for the implementation of the system, the tools available to support it still have a great deal of maturing to do. Although the compiler used on the Encore was validated, it soon became obvious that development in a parallel environment is as dependent on the correct operation of the run-time system as it is on the compiler generating valid executable code. Several problems were encountered with the Encore run-time system in the areas of stack checking, exception propagation, and task allocation.

Because the Ada language standard does not cover run-time system implementation, there is no validation capability available for run-time system operation. Therefore, as was the case with previous languages, the user is dependent on vender testing to ensure a valid system. A method of central validation was one of the major reasons Ada was developed, and serious consideration should be

given to adding run-time system standards to the current language standard.

The development tools available for this project were also inadequate. The compilers used were slow, especially toward the end of the project when long lists of dependent packages had to be recompiled because of minor code changes. The symbolic debugger was next to useless in a multitasking environment and did not support concurrent multitasking at all. There were also no tools available to monitor the execution of the parallel processes, making task analysis very difficult. Finally, the documentation available on the concurrent aspects of the run-time system was very sparse.

Parallel processing and the Ada language do hold great promise for developing complex real-time and ground-based systems. The successful implementation of this project proves that concurrent Ada is a reality and provides a number of advantages not found in other languages. However, much research remains to be done in this area. Software engineering methods must be updated to meet the new challenges of concurrent Ada development. Also, advances are needed in Ada tools and automated parallel design tools before any software project the size of that required by SDI can be attempted.

References

[1] Booch, Grady. *Software Engineering with Ada.* Menlo Park: Benjamin/Cummings Publishing Company, Inc, 1983.

[2] Cornhill, Dennis. "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets,"

[3] Cornhill, Dennis. "Partitioning Ada Programs for Execution on Distributed Systems," *IEEE Proceedings of the International Conference on Data Engineering.* 364-370. New York: IEEE Press, 1984.

[4] Encore Computer Corporation. *Multimax Technical Summary* 726-01759 Rev D. March, 1987.

[5] Encore Computer Corporation. *VADS Release Notes Concurrent Ada (B 1.0)* January 18, 1988.

[6] Hutcheon, A. D. and A. J. Wellings. "Ada for Distributed Systems," *Computer Standards and Interfaces, 6*: 71-81 (1987).

[7] Lemanski, Walter J. "Parallel Ada Implementation of a Multiple Model Kalman Filter Tracking System: A Software Engineering Approach," MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, March 1989.

[8] Maybeck, Peter S. and Daniel E. Mercier. "A Target Tracker Using Spatially Distributed Infrared Measurements" *IEEE Transactions on Automatic Control, AC-25 No. 2*: 222-225 (April 1980).

[9] Maybeck, Peter S. and Steven K. Rodgers. "Adaptive Tracking of Multiple Hot-Spot Target IR Images," *IEEE Transactions on Automatic Control, AC-28 No. 10*: 937-943 (October 1983).

[10] Maybeck, Peter S. and Robert I. Suizu. "Adaptive Tracker Field-of-View Variation Via Multiple Model Filtering" *IEEE Transactions on Aerospace and Electronic Systems, AES-21 No. 4*: 529-539 (July 1985).

[11] Nielsen, Kjell W. and Ken Shumate. *Designing Large Real-Time Systems with Ada.* New York: Multiscience Press, Inc, 1988.

[12] Quinn, Micheal J. *Designing Efficient Algorithms for Parallel Computers* New York: McGraw-Hill, Inc, 1987.

[13] SofTech, Inc. *Designing Real-Time Systems in Ada* Final Report, 1986 (AD-A169687).

[14] Tobin, David M. and Peter S. Maybeck. "Substantial Enhancements to a Multiple Model Adaptive Tracking Algorithm for a High Energy Laser Weapon System," *Proceedings of the IEEE Conference on Decision and Control.* 2002-2011. New York: IEEE Press, 1987.