PARALLEL COMPUTING FOR UNDERGRADUATES

Colgate University
June 22-24, 1994

Sponsored by

Colgate University

Undergraduate Parallel Computing Consortium
(UparCC)

The National Science Foundation
(Grants USE-9154145 and USE-9156031)

Editor
Christopher H. Nevison

Teaching Parallel Computing without a Separate Course [1]

Marcin Paprzycki and Janusz Zalewski
Dept. of Mathematics and Computer Science
The University of Texas of the Permian Basin
Odessa, TX 79762-0001
(915) 552-2258/2260
paprzycki_m,zalewski_j@gusher.pb.utexas.edu

Abstract. *This paper discusses an approach to introduce parallel computing into the CS curriculum. The basic assumptions are outlined, followed by a discussion of topics and their implementation in core courses. Several examples of exercises, demos, and assignments are given.*

1 Introduction

Our objective in teaching parallel computing is to produce CS majors who are thinking in terms of parallelism when developing applications, taking the concept of parallelism as a system development paradigm. This goal requires a change in attitude toward computations and cannot be reached by including just one or two courses on parallel computing in the curriculum. It requires a comprehensive approach to change the way students start attacking computational problems as being inherently parallel.

This is in response to the definitive advances in high performance computing (sequential and parallel) taking place over the last couple of years, that led to the formulation of the so called Computational Grand Challenges [27]. Solutions to these problems will require substantial progress in at least five areas:

- mathematical understanding of the problems posed,
- the development of new algorithms,
- improvement in software construction methods and tools,
- development in implementation techniques,
- development of new hardware.

One of the important issues that the educational community needs to address is to prepare our students for these challenges. We argue that substantial changes in the computer science education are necessary to properly prepare the next generations of graduates.

[1]Work supported in part by a grant from ARPA, via USAF Phillips Laboratory, Solicitation No. F29601-94-K-0046.

This paper discusses one approach to meet this demand. In Section 2, basic definitions and assumptions are presented. Section 3 outlines our understanding of the contents of the curriculum. In Section 4, the proposed implementation is discussed, followed by detailed examples in Section 5 and a conclusion in Section 6.

2 Basic Definitions and Para- digms

2.1 Definitions

Before we start explaining our approach, we think it is important to define the domain of discourse, that is, the subject of *parallel computing*, in contrast to related areas, such as *concurrent programming*, *distributed processing*, or *computer networking*. As we teach courses on all these topics, we have to understand their boundaries. Their overlap, although unavoidable, should be kept to a minimum.

The definitions we are using are presented below, based on those given in the IEEE Standard 610.12 Glossary on Software Engineering Terminology [16]:

Concurrent. Pertaining to the occurrence of two or more activities simultaneously within the same interval of time.

Parallel. Pertaining to the occurrence of two or more activities simultaneously at the same time instant.

Distributed. Pertaining to the occurrence of two or more activities simultaneously at different points in space.

¿From the above definitions it stems that parallel and distributed systems are concurrent, however parallel and distributed systems are not identical. The practical distinction between parallel and distributed systems is such that, if the communication time between processors is negligible then the system is parallel (if the communication time is not negligible, say comparable with the computation time, then the system is distributed). Quite naturally, this distinction has nothing to do with shared memory vs. message passing models. There are shared memory computer systems which are distributed, as well as message passing systems which are parallel. *Computer networks* fall into the distributed systems category.

It is worth noting, in particular, that for distributed systems, the notion of the same time instant does not exist, mostly because of the difficulty to prove that two events occurring at two different locations separated by a meaningful distance happened at the same time. This is difficult because distributed processors have different execution times, in contrast to parallel processors which have the same execution time line, by definition (due to the negligible communication time).

These definitions help us to define the subject more clearly. We do not claim their general applicability, though.

2.2 Paradigms

There exists a variety of approaches to teaching parallel computing. Most of them assume that a special course will be devoted to a parallel computing subject area. Such a course (or courses) can be concentrated around vector-processor-based high performance computing, an overview of software packages available on high performance computers, a senior level undergraduate course or a course sequence [28, 33]. The literature on parallel computing education has been very rich and growing over the last couple of years [1, 5, 7, 9, 11-15, 17-20, 22, 25-26, 28-37].

All those approaches have their advantages and disadvantages. The primary disadvantage of most of them is the fact that this is just one more course to be offered. There are two possible ways of handling this. Either this course will be an elective and thus not all students will take it or it will be a mandatory course. In the latter case there is a serious problem that needs to be addressed: that of the total number of semester credit hours to be part of the degree.

At our school, to graduate with a degree in computer science, a student needs to take courses in a number of fields. There is a general education requirement (partially imposed by the legislature, partially by the Texas Board of Higher Education and partially internally), which consists of approximately 53 semester credit hours (SCH). The mathematics support adds additional 12 SCH's.

The courses in the degree program (two introductory courses, programming language, four core courses, four electives and a senior research project) combine to a total of 39 SCH's. Adding to it 18 SCH's for the academic minor makes about 122 SCH's. This is under a rather unrealistic assumption that the student is immediately ready to take Calculus I without taking Precalculus first. We are quite close to what in the state of Texas is the maximum number of SCH's allowed for a degree. Adding one more course would therefore most likely require to remove something from the current list.

Instead of going the way of adding new requirements we are suggesting a different approach. This approach is based on the assumption that a single course is not enough in itself and that deeper changes in the CS curriculum are required. Rather than introducing a new course, elements of parallel computing should become, as far as possible, a part of each and every course that is currently taught. Such an approach has been already proposed in the literature and implemented in practice [18, 19], however, with slightly different assumptions in mind.

In summary, our paradigm to reach the goals of parallel computing education can be characterized by the following:
- start as early as we can in the CS curriculum, possibly in CS1 or CS2,
- teach as much as we can in core courses, in a breadth-first manner.

3 Topics – Hierarchical Approach

3.1 Computing Curricula '91

If one decides to introduce parallel computing, the next immediate question is what ma terial to include. The usual source of such information and general guidance for compute science programs, the ACM/IEEE Computing Curricula '91 [38], is not very helpful in thi respect. All it contains is the following suggestion on Parallel Computing, in the sectior *Advanced/Supplemental Material:*

Parallel and Distributed Computing

Topic Summary: This topic involves the design, structure, and use of systems having interacting processors. It includes concepts from most of the nine subject areas of the discipline of computing. Concepts from AL, PL, AR, OS, and SE are important for the basic support of parallel and distributed systems, while concepts from NU, DB, AI, and HU are important in many applications.

Subtopics include concurrency and synchronization; architectural support; programming language conctructs for parallel computing; parallel algorithms and complexity; messages vs. remote procedure calls vs. shared memory models; structural alternatives (e.g., master-slave, client-server, fully distributed, cooperating objects); coupling (tight vs. loose); naming and binding; verification, validation, and maintenance issues; fault tolerance and reliability; replication and avoidability; security; standards and protocols; temporal concerns (persistence, serializability); data coherence; load balancing and scheduling; appropriate applications.

Suggested Laboratories. Programming assignments should ideally be developed on a multiprocessor or simulated parallel processing architecture.

Prerequisites: AL9, AR6, AR7, OS (all), PL11, PL12, SE3, SE5.

3.2 Our Approach

In recognition of the fact that all technological progress, not only in parallel computing, is driven by applications, our approach starts at the top level and includes five relatively well separated layers of knowledge of parallel computing:

- understanding parallel applications
- parallel algorithms
- methodologies for parallel software construction
- implementation techniques
- parallel hardware architectures.

It is important to note that across this selection of topics, we adopt another paradigm: use and demonstrate, as far as possible, performance measures. Below we discuss the most important aspects and emphasis on each level.

1. Applications. They should demonstrate to the students the usefulness of parallel computing in a sense that there are no other methods to solve certain problems or that there are no such efficient methods. The most successful applications of parallel computing, in such fields as astronomy, molecular biology, quantum chemistry, fluid dynamics, theoretical physics, structural modeling, atmospheric and ocean research, etc. can be overviewed, however, they are too big to be successfully demonstrated. These large examples can be only described. Medium-size examples are needed, and we are taking them from the fields of real-time systems and numerical computations.

2. Algorithms. The need and demand of applications, first of all, enforce the development of new algorithms. This topic is especially rich in good examples and several classes of algorithms can be distinguished for solving typical problems, such as:

- parallel selection and searching, merging and sorting algorithms,
- parallel graph algorithms,
- parallel numerical algorithms,
- computational geometry and image processing,
- parallel combinatorial algorithms,
- parallel optimization,
- neural network and genetic algorithms.

What we emphasize to include in our curriculum is sequential and parallel sorting, sequential and parallel searching, and parallel solutions to the optimization problems (such as dynamic programming). An essential part of this topic is how the sequential algorithms can be improved by their parallelization. To prove this, algorithm complexity and the big-O notation should be enforced.

3. Methodologies. The transition from the knowledge of algorithms to the practice of software development methodologies should be the key issue in this topic. In general, a good software development methodology, for developing either traditional sequential or parallel software, should include three aspects: the method, techniques, and tools. The *method* is a set of paradigms and a notation to express and specify the intended software properties and their relationships. *Techniques* are mathematical transformations that support the paradigms and enable software development in a systematic and rigorous manner, and *tools* are software packages that help in automatic development.

For traditional sequential systems development, there are several methods and techniques, well described in the literature and supported by tools. At the same time (to the best of our knowledge) no such comprehensive methodology exists that would comprise all three necessary components and aim parallel systems. Regarding methods and techniques, a promising approach is represented by Petri nets [3], however, there seems to be a lack of commercial tools supporting this approach. Among the environments which offer support for most phases of the software development cycle, although their theoretical basis is unclear, there are two worth mentioning:

- CODE/ROPE [4]
- PPSE [23].

The major problem with adopting traditional methodologies to developing parallel systems is that they support exclusively functional (control) parallelism and not data parallelism, necessary in parallel software development. For this reason, well established development methodologies for real-time systems, for example, structured or object-oriented, cannot be used directly.

4. Implementation techniques. Implementation techniques form one of the best developed segments of parallel computing. They comprise languages for parallel computations and respective operating system constructs. Based entirely on relatively well developed theory of concurrency, they define models of concurrent and parallel computations and define language primitives most suitable from the point of view of such models. Depending on the assumed model of concurrency, whether shared memory or message passing, one can use in teaching several primitives designed for concurrent programming: semaphores, signals, event flags, critical regions, barriers, monitors, mailboxes, rendezvous, remote procedure calls, and others.

There are several languages used for implementation of parallel systems, from traditional languages, such as Fortran and C, through Ada, Occam, Linda, Modula-P, Concurrent C++, Erlang, and functional, as well as, logic programming languages. Our approach to the language issue may be controversial but is based on an extensive industrial experience: it does not matter very much which language we use – what is important are concepts. One part of this attitude is mixing languages: in practice there are very few meaningful applications (if any) written exclusively in one language. Therefore programming in a variety of languages should be an essential component of computer science education.

5. Architectures. Hardware architectures for parallel computations comprise a variety of approaches, mostly following the perpetual Flynn classification. An important issue which needs to be addressed here is a distinction between shared memory (tightly coupled) multiprocessors and message passing (loosely coupled) multiprocessors. Another crucial problem is the type of interconnection, which includes three principal categories:

- bus
- point-to-point links
- crossbar switches.

Our approach to teaching this subject, again based on experience, favors standardization [42]. Backplane bus architectures and their evolution towards solutions such as SCI (Scalable Coherent Interface) or HIPPI (HIgh Performance Parallel Interface) are therefore emphasized, although other architectures are not eliminated.

This broad coverage effectively means that we attempt to teach all aspects of parallel computations, rather than focus on a single aspect such as parallel algorithms or parallel architectures. Following our definition of parallel computing, we do not include topics such as communication, fault tolerance, etc., which fall into distributed processing.

4 Implementation

In this section we discuss the implementation of the approach outlined above, in the core courses at our institution. We emphasize the initial courses, firstly because they are the most important vehicle to spread the ideas outlined above, and secondly because we have not done much for upper-division courses yet. The key issue in the implementation of the program is how to map basic concepts of parallel computing onto the course sequence.

4.1 Lower-division Courses

1. Computer Science I
Our teaching micro-paradigm in this course is the software engineering approach: Design/Implement/Test, without introducing parallelism explicitly. Therefore we cover preparatory concepts which we consider important from the point of view of parallel computations: local/global variables, information hiding (ADTs), separate compilation, vector and matrix computations. The idea of parallelism can be discussed when introducing the concept of an algorithm.

The idea of a computer with more than one processor has to be introduced. A good example to convince students that the future of computing will consist of multiple processor environments executing user applications is what can be already observed in top-of-the-line PC's, where we have the main processor, a video processor, the sound processor, floating point processor and others.

2. Computer Science II
The emphasis is on parallelization of elementary searching and sorting, for example, following practical applications [2, 8]. Optimization algorithms are introduced as those of extreme practical importance. The concept of load balancing is briefly discussed.

Matrix multiplication, as a computation-intensive application especially suitable for performance evaluation, is one of the major topics. On shared-memory multiprocessors, rescheduling algorithms work well and exhibit good speed-up characteristics with an increase in the number of processors. On message-passing architectures, matrices are decomposed into subblocks and multiplied, and a regular communication pattern between the processing elements helps minimize the overheads.

3. Discrete mathematics
Includes basic concepts that need to be used when designing and analyzing parallel algorithms: introduction to Petri nets, introduction to cellular automata, graph theory, combinatorics.

4. Programming Languages

(or Operating Systems)
This is the place to teach implementation techniques of parallel computing, either on the language or on the operating system level. There exists a number of possible selections that can be categorized as follows:

- standard languages with parallel costructs built in, such as Ada, Fortran 90 and HP Fortran
- languages with parallel extensions, such as C
- languages designed specifically for parallel computing, such as Linda and others [6].

The implementation issues, such as multiple processes or tasks, and selected interprocess communication constructs, can be covered in any of those languages. The decision on whichever language is selected may be based on the anticipated needs to do the parallel programming projects in upper-division courses.

4.2 Upper-division Courses

We are not prepared, yet, to give a comprehensive characteristics and discuss the four remaining courses of our core. A preliminary description of the basic issues of parallel computing which can be included, is given below.

5. Computer Architecture
Instruction-level parallelism, cache and cache coherence problems, bus arbitration, parallel I/O. Hardware lab is essential for this type of course.

6. Data Structures and Algorithms
Advanced concepts of parallel algorithms, introduction to performance analysis of parallel algorithms, formal analysis of time/space complexity of parallel algorithms, analysis of problem scaling. An essential component of this course is comparative studies of parallel algorithms.

7. Software Engineering, and
8. Senior Research Project
Parallel/concurrent software development methodologies, use of tools.

5 Examples

What we consider crucial to the success of such an approach is a set of carefully selected examples (demos), exercises, assignments, and team projects from the cross-section of the discipline. Our former experience with other courses proved that this way of gradual introduction of new concepts significantly enforces their understanding [40].

5.1 Demos and Exercises

A selection of demos for concurrent and parallel programming is based on the idea discussed elsewhere [40]. The concept of demonstrating more realistic applications [39] has been also successfully tested in practice. Below we present a sample of vector/vector exercises to be used in CS I or CS II.

Example 1

Operation description:
Rank one update of *axpy* – a scaled vector added to another vector; it is assumed that x and y are vectors of length n.

Basic algorithm (x and y are initialized):

```
      DO 10 I = 1,N
10          X(I) = X(I) + a*Y(I).
```

Discussion topics:

1. How can the work be divided on a computer with more than one processor?
2. How would you measure the success of such division of work?
3. If you have p processors and the vector is of length n how are you going to divide the work?
4. What to do if n is not divisible by p?
5. What will happen if one of the processors has more/less work that the other processors?
6. Concepts of pre-scheduling and self-scheduling.
7. Will it matter if there was one global memory or each processor would have individual memory?

Exercise:
In class (closed lab) write a parallel program; show that it does work; show that for more than one processor there is a time reduction.

Example 2

Operation description:
Calculate the dot-product of x and y, both vectors of length n.

Basic algorithm (x and y are initialized):

```
      TEMP = 0.0
      DO 10 I = 1,N
10          TEMP = TEMP + X(I)*Y(I).
```

Discussion topics:

1. What is the difference between this example and the *axpy* example?
2. How would you divide work?
3. Can self scheduling be achieved and how (what mechanism is necessary)?
4. What will happen if p is large in comparison to n?
5. What will be the effect of having a computer with local memories?

Exercise:
Write a program that performs the operation in parallel; discuss it; show that it works.

5.2 Assignments and Projects

Team projects are very important from the software engineering point of view which we advocate, and can be implemented as suggested in [41]. However, we are not ready yet to test this idea in a classroom.

On the issue of assignments, we adopted two basic ways of conducting them. One very efficient way is to extend exercises to a full assignment, same for the entire class. Another way, that is more effective but involves an instructor to a much larger extent, is to assign a problem to solve individually per student. An example of the individual assignment in concurrent programming, to control access to a one-way tunnel, written in Ada, is presented in the Appendix. It has been written by a CS II student with an extensive help of an instructor. Sample assignments for the algorithms course are also available, for example [24]. Below we present one way of extending exercises mentioned above to a full assignment.

Sample Assignment:
For the *axpy* and dot product exercises, run experiments for various values of n and p (system dependent numbers) and observe the performance.

Questions to be addressed by the students:

1. What happens to performance for fixed p and increasing n?
2. What happens to performance for fixed n and indreasing p?
3. How can you explain what you see?

6 Conclusion

Our approach to teaching parallel computing without a separate course can be characterized as follows:

- Understand parallel computing
- Define your teaching objective
- Select a range of topics

- Map topics onto courses.

In this paper we do not address the issue of the method (how to teach), except of mentioning the demos/exercises/assignments/project sequence. The fundamental question, in this respect, is obviously the existence and usefulness of a parallel computing laboratory. Such a lab is currently being developed. Its major function will be to serve as a teaching aid in developing the concept of Closed Labs.

One interesting experience, so far, was that we found very useful the idea of *recursive concepts* [38]. For instance, in parallel computing, one can use the notion of multiple entities competing for a single resource, on virtually all levels of a hierarchy of topics, in various forms, such as: job partitioning (algorithm level), resource allocation (development level), task scheduling (implementation level), bus arbitration (architecture level). Two other notions which can be used this way, across the entire parallel computing curriculum, are: performance evaluation (of algorithms, development methods, implementations, and architectures) and heterogeneity (mixing programming environments, mixing languages, and mixing architectures).

First response from the students, though very limited, is rather positive. Most of them are excited about being able to use and develop parallel applications to meet the market demands. We are aware, however, that the implementation of the entire project will undergo a very lengthy and painful process.

One interesting question is: which textbook to use if parallel computing topics are spread over the entire curriculum? In our opinion, using a textbook in a classical way does not make much sense, and we are seeking less traditional ways of enhancing each particular course. One very important method of doing this is to use multimedia and similar type of course materials [10, 21]

It would be also interesting to compare our approach and results with other comprehensive attempts to integrate parallel computing with the entire CS curriculum, such as the one published in [18, 19].

References

[1] Bachelis G., B. Maxim, Tutorial: Introducing Parallel Algorithms in Undergraduate Computer Science Course, SIGCSE Bulletin, Vol. 22, No. 1, p. 255, 1990

[2] Boykin J. et al., Programming under Mach, Addison-Wesley, Reading (MA), 1993

[3] Braunl T., Parallel Programming. An Introduction. Prentice Hall, Englewood Cliffs (NJ), 1993

[4] Browne J.C., M. Azam, S. Sobek, CODE: A Unified Approach to Parallel Programming. IEEE Software, Vol. 6, No. 4, pp. 10-18, August 1989

[5] Butler R.M., R.E. Eggen, S.R. Wallace, Introducing Parallel Processing at the Undergraduate Level, SIGCSE Bulletin, Vol. 20, No. 1, pp. 63-67, 1988

[6] Cheng D.Y., A Survey of Parallel Programming Languages and Tools, Report RND-93-005, NASA Ames Research Center, Moffett Field (CA), March 1993

[7] Duckworth R.J., Introducing Parallel processing Concepts Using the MASPAR MP-1 Computer, SIGCSE Bulletin, Vol. 26, No. 1, pp. 353-356, March 1994

[8] Ellis C.S., Concurrent Search and Insertion in AVL Trees, IEEE Trans. on Computers, Vol. 29, No. 9, pp. 811-817, September 1980

[9] Fisher A.L., T. Gross, Teaching the Programming of Parallel Computers, SIGCSE Bulletin, Vol. 23, No. 1, pp. 102-107, 1991

[10] Gloor P.A. et al. (Eds.), Parallel Computation – Practical Implementation of Algorithms and Machines (CD-ROM). Telos/Springer-Verlag, Santa Clara (CA), 1994

[11] Guha R.K., J. Hartman, Teaching Parallel Processing: Where Architectures and Language Meet, Proc. IEEE Conf. Frontiers in Education, 1992

[12] Hartman J., D. Sanders, Data Parallel Programming: A Transition from Sequential to Parallel Computing, SIGCSE Bulletin, Vol. 25, No. 1, pp. 96-100, 1993

[13] Hartman J., D. Sanders, Teaching a Course in Parallel Processing with Limited Resources, SIGCSE Bulletin, Vol. 23, No. 1, pp. 97-101, 1991

[14] Hintz T., Introducing Undergraduates to Parallel Processing, IEEE Trans. on Education, Vol. 36, No. 1, pp. 210-213, 1993

[15] Hyde D.C., A Parallel Processing Course for Undergraduates, SIGCSE Bulletin, Vol. 21, No. 1, pp. 170-173, 1989

[16] Institute of Electrical and Electronics Engineers, IEEE Std 610.12 Glossary of Software Engineering Terminology. IEEE, New York, 1990

[17] Jipping M.J., J.R. Toppen, S. Weeber, S., Concurrent Distributed Pascal: A Hands-on Introduction to Parallelism, SIGCSE Bulletin, Vol. 22, No. 1, pp. 94-99, 1990

[18] John D.J., Integration of Parallel Computation into Introductory Computer Science, SIGCSE Bulletin, Vol. 24, No. 1, pp. 281-285, 1992

[19] John D.J., NSF Supported Projects: Parallel Computation as an Integrated Component in the Undergraduate Curriculum in Computer Science, SIGCSE Bulletin, Vol. 26, No. 1, pp. 357-361, March 1994

[20] Katsinis C., The Development of a Multi-Processor Personal Computer in a Senior Computer Design Laboratory, SIGCSE Bulletin, Vol. 26, No. 1, pp. 349-351, March 1994

[21] Kennedy K. et al., Parallel Computation: Practice, Perspectives and Potential. CRPC Short Course (7 videotapes). California Institute of Technology, Pasadena (CA), 1994

[22] Kitchen A.T., N.C. Schaller, P.T. Tymann, Game Playing as a Technique for Teaching Parallel Computing Concepts, SIGCSE Bulletin, Vol. 24, No. 3, pp. 35-38, 1992

[23] Lewis T.G., W.G. Rudd, Architecture for Parallel Programming Support Environments. Proc. COMPCON '90, pp. 589-594, IEEE Computer Society Press, Los Alamitos (CA), 1990

[24] Litwin W., Y. Sagiv, K. Vidyasankar, Concurrency and Trie Hashing, Acta Informatica, Vol. 26, pp. 597-614, 1989

[25] Luque E., R. Suppi, J. Sorribes, A Quantitative Approach for Teaching Parallel Computing, SIGCSE Bulletin, Vol. 24, No. 1, pp. 286-298, 1992

[26] Meredith M.J., Introducing Parallel Computing into the Undergraduate Computer Science Curriculum, SIGCSE Bulletin, Vol. 24, No. 1, pp. 187-191, 1992

[27] Messina P., T.Sterling (Eds.), System Software and Tools for High Performance Computing Environments, SIAM, Philadelphia (PA), 1993

[28] Miller R., The Status of Parallel Processing Education, Draft Report, SUNY Buffalo, September 1993

[29] Mims T., A. Hoppe, Utilizing a Transputer Laboratory and Occam2 in an Undergraduate Operating Systems Course, SIGCSE Bulletin, Vol. 23, No. 1, pp. 317-323, 1991

[30] Nevison C., An Undergraduate Parallel Processing Laboratory, SIGCSE Bulletin, Vol. 20, No. 1, pp. 68-72, 1988

[31] Nevison C. et al. (Eds.), Laboratories for Parallel Computing, Jones and Bartlett Publishers, Boston (MA), 1994

[32] Paprzycki M., Incorporating High-Performance Computers into Mathematics Curriculum, Proc. Fifth Ann. Conf. on Technology in Collegiate Mathematics, pp. 862-868, Addison-Wesley, Reading (MA), 1993

[33] Paprzycki M., J. Zalewski, Introduction to Parallel Computing Education, Journal of Computing in Small Colleges, Vol. 9, No. 5, pp. 85-92, May 1994

[34] Rifkin A., Teaching Parallel Programming and Software Engineering to High School Students, SIGCSE Bulletin, Vol. 26, No. 1, pp. 26-30, March 1994

[35] Posch R., F. Pucher, M. Welser, Using a Transputer Cluster in a Classrom Environment, Computer Communications, Vol. 16, No. 3, pp. 192-196, March 1993

[36] Sanders D. J. Hartman, Getting Started with Parallel Programming, SIGCSE Bulletin, Vol. 22, No. 1, pp. 86-88, 1990

[37] Torsone C.M., Introducing Parallel Programming to a Programming Language Concepts Course, Journal of Computing in Small Colleges, Vol. 9, No. 2, pp. 66-70, February 1993

[38] Tucker A.B. (Ed.), Computing Curricula '91. Report of the ACM/IEEE-CS Joint Curriculum Task Force, ACM/IEEE, New York, 1991

[39] Wann K.C., J. Zalewski, Scheduling Messages in Real Time with Application to the SSC Message Broadcast System. IEEE Trans. on Nuclear Science, Vol. 41, No. 1, pp. 213-215, February 1994

[40] Zalewski J., A Real-Time Systems Course Based on Ada, Proc. 7th Annual Ada Software Engineering Education and Training Symposium (ASEET), pp. 25-49, Monterey, CA, January 12-14, 1993

[41] Zalewski J., Cohesive Use of Commercial Tools in a Classroom, Proc. 7th SEI Conf. on Software Engineering Education, pp. 65-75, San Antonio, TX, January 5-7, 1994, J.L. Diaz-Herrera (Ed.), Springer-Verlag, Berlin, 1994

[42] Zalewski J. (Ed.), Advanced Multi- microprocessor Bus Architectures. A Tutorial. IEEE Computer Society Press, Los Alamitos (CA), 1994 (in print)

Appendix

```
---------------------------------------------------------------------------
-- Program Tunnel.  Design by: Teresa Spraggins                          --
-- University of Texas-Permian Basin, April 28, 1994                     --
-- This program keeps track of a one way tunnel's traffic light.         --
-- It only allows N cars (constant below) to be in the tunnel at a time. --
-- Loosely follows the specification (Exercise 1.3, p. 22) from:         --
-- Concurrent Programnming, by Tom Axford, John Wiley and Sons, 1991     --
---------------------------------------------------------------------------
-- Global declarations
   MaxCars : constant INTEGER := 6;  -- only that many car tasks may exist
   N_Vehicles : INTEGER := 0;        -- protected by the Semaphore
---------------------------------------------------------------------------
--                         SEMAPHORE TASK                                --
---------------------------------------------------------------------------
-- This task will protect the variable N_Vehicles (of cars in a tunnel)

task body Semaphore is
begin
   loop
      accept  Lock;
      accept  UnLock;
   end loop;
end Semaphore;

---------------------------------------------------------------------------
--                         DISPLAY SEMAPHORE TASK                        --
---------------------------------------------------------------------------
-- This task will protect the screen against simultaneous writes

task body Display_Sema is
begin
   loop
      accept  Lock;
      accept  UnLock;
   end loop;
end Display_Sema;

---------------------------------------------------------------------------
--                         CAR ENTRANCE TASK                             --
---------------------------------------------------------------------------
-- This task records all vehicles ENTERING the tunnel.
```

```
 task body Entrance_Process is
 begin
    loop
       accept Car_Entering;

       Semaphore.Lock;
       N_Vehicles := N_Vehicles + 1;
       Semaphore.UnLock;
    end loop;
 end Entrance_Process;

-------------------------------------------------------------------------
--                          CAR EXIT TASK                              --
-------------------------------------------------------------------------
-- This task records all vehicles LEAVING the tunnel.

task body  Exit_Process is
begin
   loop
      accept Car_Exiting;

      Semaphore.Lock;
      N_Vehicles := N_Vehicles - 1;
      Semaphore.UnLock;
   end loop;
end Exit_Process;

-------------------------------------------------------------------------
--                          CAR RUNNING task                           --
-------------------------------------------------------------------------
-- This task will display the car running through the tunnel

task body Car_Running is

-- This procedure does the job
   procedure  Display_Car is
      Temp : INTEGER := Random mod 4;
   begin
 if Temp = 0 then   -- Car of first type
           for I in 1..69 loop
              Display_Sema.Lock;
              Put_Truck(I, 20);    -- proceduref from
              Blank_EX(I-1, 20);  -- Screen package
              Display_Sema.UnLock;
           end loop;
```

```
  end if;

    -- There is three more loops for other types of cars displayed that way
    -- ...

         Exit_Process.Car_Exiting;

         for F in 69..71 loop   -- to drop the car from screen
            Display_Sema.Lock;
            Put_Blank(F, 20);
            Display_Sema.UnLock;
         end loop;

   end Display_Car;

begin  -- of Car_Running tasks
   Display_Car;
end Car_Running;

-------------------------------------------------------------------------------
--                        TRAFFIC LIGHT PROCESS                              --
-------------------------------------------------------------------------------
-- This task will control the traffic light.

task body Traffic_Light is
   X : INTEGER;
   N : constant := 4;   -- Maximum number of cars allowed in the tunnel
   -- Define GREEN_LIGHT and RED_LIGHT procedure
begin   -- Traffic_Light process
    loop
       Semaphore.Lock;
       X := N_Vehicles;  -- cannot be unlocked directly after this assign
                         -- because N_Vehicles may change before the light
                         -- is changed based on current N_Vehicles
       if (X >= N) then
          Red_Light;
       else
          Green_Light;
          delay DURATION(N);      -- to separate cars on screen
          GENERATE_CARS.GoAhead;  -- only after the light has turned green
       end if;                    -- a new car can be launched
       Semaphore.UnLock;
    end loop;
END Traffic_Light;
```

```
------------------------------------------------------------------------
--                          GENERATE CARS task                        --
------------------------------------------------------------------------
-- This task randomly adds cars attempting to enter the tunnel.
task body Generate_Cars is
   Ind : Integer;
   Counter : Integer := 0;
begin
   -- Initialization: display the tunnel and set light to GREEN
   loop
         Counter := Counter + 1;  -- theoretically may overflow
         delay Duration(Float(Random)/8.0 + 0.2);

         Ind := Counter MOD MaxCars;
         if Ind = 0 then
            Ind := MaxCars;
         end if;

         if Counter > MaxCars then
            Dispose(RunningCars(Ind));          -- Remove a car task from memory
         end if;

         accept GoAhead;                        -- Permission to generate a new
         Entrance_Process.Car_Entering;         -- car to enter the tunnel:
         RunningCars(Ind) := new Car_Running;   -- Here!!  Starts displaying.
   end loop;
end Generate_Cars;
------------------------------------------------------------------------
```