Rosalind L. Ibrahim (Ed.)

Software Engineering Education

8th SEI CSEE Conference
New Orleans, LA, USA
March 29 - April 1, 1995
Proceedings

Springer

Parallel and Distributed Computing Education: A Software Engineering Approach [1]

Marcin Paprzycki, Ryszard Wasniowski[2], Janusz Zalewski[3]

Dept. of Math and Computer Science
University of Texas–Permian Basin
Odessa, TX 79762
paprzycki_m@gusher.pb.utexas.edu

[2]Computer Science Dept.
Central Missouri State University
Warrensburg, MO 64093
rwasniowski@acm.org

[3]Computer Science Dept.
Embry-Riddle Aeronautical University
Daytona Beach, FL 32114
zalewski@db.erau.edu

Abstract. *This paper discusses an approach, based on software engineering principles, to introduce parallel and distributed computing into the CS curriculum. The basic assumptions are outlined, followed by a discussion of topics and their implementation in core courses. Innovations in the teaching method are also presented. Several examples of exercises and assignments are given.*

1 Introduction

In recent years, parallel and distributed computing have been considered to be the most promising solution to the computational requirements of the future. Significant advances in parallel and distributed algorithms and architectures, including those in communication technologies, have shown the enormous potential of such computational techniques in a variety of practical problems. However, most of the research efforts have concentrated either upon computational models, parallel/distributed versions of algorithms, or machine architectures. Relatively little attention has been paid to software development environments or program construction techniques that are required to translate algorithms into operational programs. This aspect is becoming more important as parallel/distributed processing progresses from the research laboratories to applications.

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

One of the reasons for a relatively slow pace of introducing parallel and distributed computing technology into real-life applications is a shortage of an adequately qualified workforce. In our opinion, this is mostly due to a gap in computing curricula that do not put enough attention to software engineering principles.

Our objective in teaching parallel and distributed computing is to produce CS majors who are thinking in terms of concurrency when developing applications, taking the concepts of parallelism and distribution 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 and/or distributed computing in the curriculum. It requires a comprehensive approach to change the way students start attacking computational problems as being inherently parallel.

From the software engineering viewpoint, solutions to current and future application problems will require comprehensive understanding of at least five areas of parallel and distributed computing:

- mathematical models of the problems posed
- inherent parallelism of new algorithms
- methods and tools for software construction
- principles of implementation techniques
- new hardware architectures and interconnects.

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 educate the next generations of graduates. This is especially important in a view of the most recent advances in communication technologies.

This paper discusses one approach to meet this demand and summarizes our experiences in using software engineering principles to teach parallel and distributed computing to undergraduates. In Section 2, basic definitions and assumptions are presented. Section 3 outlines our understanding of the contents of the curriculum, based on software engineering principles. In Section 4, the sample implementation is described, followed by detailed examples in Section 5, a discussion of experiences in Section 6, and a conclusion in Section 7.

2 Basic Definitions and Paradigms

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/distributed computing*, in contrast to related areas, such as *concurrent programming* 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 [5]:

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 of computation. There are shared memory computer systems that are distributed, as well as message passing systems that are parallel.

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 to prove, because distributed processors have different execution times, in contrast to parallel processors that have the same execution time line, by definition (due to the negligible communication time).

Distributed systems are sometimes further divided into two different categories: computing clusters and computer networks, depending on the existence of a common goal of a computation. Thus, physically the same computer network can be used for cluster computing, if it provides means to meet a common objective – say, to solve a certain computation-intensive task. Meeting this objective usually requires a special-purpose software, in addition to general networking software.

The above mentioned definitions and distinctions help us to define the subject more clearly. They are especially useful from the educational viewpoint, we do not claim their general applicability, though.

2.2 Paradigms

There exists a variety of approaches to teaching parallel and distributed computing. The literature on parallel computing education has been especially rich and growing over the last couple of years [15]. Most of these approaches assume that a special course or a course sequence will be devoted to this 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 [11, 14]. The primary disadvantage of most of them is the fact that this is just one more course to be offered.

Instead of adding new course 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 and distributed computing should become, as far as possible, a part of each and every course that is currently taught. Such an approach, based on a software engineering view of product development, helps to develop the necessary skills from the very beginning and keep developing them throughout the entire course of study.

In summary, our paradigm to reach the goals of parallel and distributed 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
- base the introduction of the material into the curriculum on software engineering principles.

3 Curriculum Contents – A Hierarchical Approach

If one decides to introduce parallel and distributed computing into the curriculum, the next immediate question is what material to include. The usual source of such information and general guidance for computer science programs, the ACM/IEEE Computing Curricula '91 [17], is not very helpful in this respect. All it contains is the two-paragraph suggestion on Parallel/Distributed Computing in the section *Advanced/Supplemental Material*.

According to the software engineering view, to be able to develop parallel and distributed applications, our approach starts at the top level and includes five relatively well separated layers of knowledge of parallel/distributed computing:

- understanding parallel and distributed applications
- parallel and distributed algorithms
- methodologies for parallel/distributed software construction
- implementation techniques
- parallel and distributed architectures and interconnects.

Below we discuss the most important aspects and emphasis on each level.

1. Applications. They should demonstrate to the students the usefulness of parallel and distributed 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/distributed computing, in such fields as astronomy, molecular biology, quantum chemistry, fluid dynamics, theoretical physics, structural modeling, atmospheric and ocean research, etc., can be overviewed, but they are usually too big to be successfully demonstrated. These large examples can be only described. Medium-size examples are needed, which we are taking from the fields of numerical computations and real-time systems [13, 19, 20].

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, from selection and searching through neural network and genetic algorithms. What we emphasize to include in our curriculum, in addition to traditional algorithms such as searching and sorting, are parallel and distributed solutions to the optimization problems. An essential part of this topic is how the sequential algorithms can be improved by their parallelization. To prove this, algorithm complexity theory 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 on this level. 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, commercially successful, methodology exists that would comprise all three necessary components and aim parallel and/or distributed systems. Regarding methods and techniques, a promising approach is represented by Petri nets. However, despite a number of public domain tools (for example, those listed in [6]), there seems to be a lack of coherent commercial tools supporting this approach.

The major problem with adopting traditional methodologies to developing parallel/distributed 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 and distributed computing. They comprise languages for parallel/distributed computations and the respective operating system constructs. Based entirely on relatively well developed theory of concurrency, they define models of concurrent (parallel/distributed) 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 teach the use of 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, Orca, and functional, as well as, logic programming languages [2, 10]. Our approach to the language issue may be unusual but is based on an extensive industrial experience: it does not matter very much which language we use – what is important are the 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/distributed computations comprise a variety of approaches, mostly following the perpetual Flynn classification. An important issue that needs to be addressed here is a distinction between shared memory (tightly coupled) systems and message passing (loosely coupled) systems. 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 [22]. 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.

In summary, this broad coverage effectively means that we attempt to teach all aspects of parallel and distributed computations, rather than focus on a single aspect such as parallel/distributed algorithms or architectures. Following the distinction between parallel and distributed computing, topics such as communication, fault tolerance, etc., are included in courses on distributed systems.

4 Implementation

There are three major issues in the implementation of the outlined approach:

- how to map basic concepts of parallel/distributed computing onto the course sequence
- what tools to use to facilitate and enforce the absorption of the material by students

- what teaching methods to use to ensure effective transmission of knowledge.

4.1 Mapping Concepts on the Course Sequence

The authors' attempt to introduce parallel/distributed computing concepts in the core courses at one of the institutions (UTPB) is outlined below. In the presentation, 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 had extensive experience with upper-division courses yet.

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 that we consider important from the point of view of parallel/distributed computations: local/global variables, information hiding (ADTs), separate compilation, vector computations. The idea of parallelism can be discussed when introducing the concept of an algorithm.

2. Computer Science II
The emphasis is on parallelization of elementary searching and sorting, for example, following practical applications [1, 3, 9]. 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 suitable for performance evaluation, is one of the major topics.

3. Discrete Mathematics
We consider this course to be the one giving theoretical background for formal software development as well as for the design of parallel/distributed algorithms. The basic topics included are: propositional and predicate calculi, formal logic, elements of set theory, combinatorics, graph theory, introduction to Petri nets, and introduction to cellular automata.

4. Computer Architecture
This is a traditionally oriented course on introduction to computer architecture, which has to teach all basic hardware-related concepts, so not much time is left to introduce basic ideas of parallelism and distribution. If this is possible, however, we recommend to cover any of the following topics: instruction-level parallelism, cache and cache coherence problems, bus arbitration, hardware lock, and parallel I/O. Hardware lab is essential for this type of course.

5. Upper-Level Courses
Our plans, already partially implemented, include focusing on particular layers of the topics hierarchy in upper-division courses, such as:

- Programming Languages or Operating Systems (to teach particular implementation techniques)
- Data Structures and Algorithms (for teaching advanced concepts), and

- Software Engineering and Senior Research Project, as separate courses (to teach parallel/distributed software development methodologies and emphasize the use of tools).

4.2 Teaching Tools

The teaching tools we use can be divided into three major categories, based on the distinction we made about various kinds of concurrency:

- concurrent software development tools
- tools for parallel software development
- distributed software development tools.

Concurrent software development tools form the most consistent category. One particular combination has been described in a former article [21]. It is composed of high-level tools, mapping designs onto implementations in particular languages, such as Ada or C++, supported by a real-time, Unix-based, operating system running on a single-processor hardware. A number of other tools suitable for immediate use in this category are listed in Appendix 2.

There is a lack of commercially available tools for parallel software development that would cover the entire life cycle. The enormous variety of parallel architectures is another serious obstacle to come up with a cohesive set of tools for the construction of parallel systems. One possible approach, which has been implemented, includes a set of PC-based transputer boards driven by Ada/Occam-based software. There are two advantages of selecting this particular architecture and base higher-level tools on it. First, it fits well into our former experience with using tools for concurrent and real-time systems education [20, 21]. Second, such a combination of tools has been widely used in college education, and a very good text exists which covers parallel laboratories [12].

Setting up a coherent development environment for teaching distributed computing is a particular challenge due to the primary requirement of keeping it heterogeneous. Distributed computing using a cluster of workstations, called cluster computing, is becoming an attractive and viable solution. Traditionally, groups of workstations are used as a resource for a series of sequential jobs, but with the development of specialized message-passing software a cluster could also work together on a single problem. In addition, workstation clusters are an attractive and cost-effective software development platform. An application can readily be developed on a group of workstations and ported to moderately parallel machines. If the implementation of the application algorithms scales properly, then software for massively parallel machines can also be developed on the cluster platform. Workstation-based programming environments are also affordable for researchers, especially at universities with moderately sized computer centers. Below, we report on a heterogenous environment which has been adopted for use in classroom experiments on distributed computing. The PVM

project [16] is an attempt to provide a unified framework within which large programs can be developed on a collection of heterogeneous machines and make the transition from teaching sequential processing to parallel/distributed processing easy.

4.3 Cluster Computing

The term *cluster computing* means distributed computing with the use of several high-performance workstations to solve a single computation-intensive problem. An example of software suitable for this kind of computing is the PVM system, developed at Oak Ridge National Laboratory. PVM is a programming environment for the development and execution of large distributed applications that consist of many interacting but relatively independent components. It is intended to operate on a collection of heterogeneous computing systems interconnected by one or more networks. The participating processors may be scalar machines, multiprocessors, or special-purpose computers. PVM provides a straightforward and general interface that permits the description of various types of algorithms while the underlying infrastructure permits the execution of applications in a virtual computing environment that supports multiple parallel computation models.

Using PVM, large computational problems can be solved by the aggregate processing power and memory of many computers. PVM supplies the functions to start tasks and lets the computers communicate and synchronize with each other. It survives the failure of one or more connected computers and supplies functions for users to make their applications fault tolerant. Users can write applications in Fortran or C and parallelize them by calling simple PVM message-passing routines such as *pvm_send()* and *pvm_recv()*. By sending and receiving messages, application subtasks can cooperate to solve a problem concurrently.

What makes PVM especially attractive to us is its development environment, Hence. With the Hence graphics interface (also available from Oak Ridge) implemented on a workstation, a user can develop a parallel program as a computational graph. The nodes in the graph represent the computations to be performed and the arcs represent the dependencies between the computations. From this graphical representation, Hence can generate a lower level portable program, which when executed will perform the computations specified by the graph in an order consistent with the dependencies specified. Designers can visualize the problem's overall structure far more easily from these graphical representations than from textual specifications.

4.4 The Method

The method (how to teach) is crucial in implementing the approach outlined above. There are several issues we would like to stress. First, we use what is

believed to be a very effective method for developing software construction skills: the sequence demos/exercises/assignments/projects. This technique has been outlined elsewhere [15, 20, 21] and is illustrated here with a couple of examples in the next section. To illustrate it very briefly, for 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, has been presented in [15].

Another interesting experience that we found very useful is the idea of *recurring concepts* [17]. For instance, in parallel/distributed 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 we use this way across the entire computing curriculum are: performance evaluation (of algorithms, development methods, implementations, and architectures) and heterogeneity (mixing programming environments, mixing languages, and mixing architectures).

One other very important issue regarding the method is the materials for individual study. We attempt to employ new generations of multimedia techniques, departing slightly from a traditional concept of a textbook. A careful review of the market revealed enough opportunities to select a handful of good teaching aids of various kinds one can use in parallel/distributed computing education, for example:

- CD-ROMs [4]
- videotapes [7]
- electronic books [18].

5 Examples

What we consider crucial to the success of such an approach is a set of carefully selected examples including 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 [20].

The selection of such examples for the development of concurrent and parallel systems has been presented elsewhere [15, 20, 21]. The concept of demonstrating more realistic applications [19] has been also successfully tested in practice. Below we present a sample parallel sorting exercise and focus on selected exercises for distributed computing.

5.1 Sorting Exercise

Excercise objective: To teach basic issues related to the parallel sorting.

Methodology: In-class discussion followed by the code development and by the homework assignment.

Prerequisite: Knowledge of basic sequential sorting methods [8].

Discussion topics:

1. Describe the basic sorting methods that you know (It is assumed that some basic sorting methods as well as quicksort will be mentioned as a minimum.)
2. How can we parallelize the code in a multiprocessor environment with global memory? (We lead the discussion in such a way that the idea of dividing n elements to be sorted among p processors and doing the sorting by each processor independently followed by the combination of the results will be introduced.)
3. How can the results from different sorts be combined? (A multiprocessor merging should be the outcome of this discussion. The proper version of compare-split operation should be also introduced.)
4. For more advanced classes. What is the arithmetic complexity of such a sorting process? Assume that there is no communication cost involved and that the different sorting procedures are applied in the parallel phase. What are the properties of these sort procedures for fixed n and increasing number of processors; for fixed number of processors and increasing n? What is the predicted speed-up?
5. Additional topics (for higher level courses e.g. Theory of Algorithms). a) How the algorithms need to be modified for the distributed memory machine? b) How to sort the data if there are as many processors as there are data elements? c) Can we utilize an additional information about the data (e.g. data are already partially sorted). d) Effect of the communication costs on the arithmetic complexity. e) Bitonic sort and its properties, functions and their asymptotic behavior.

Class Exercise:
Implement the parallel sorting algorithm on the parallel machine available to you. Here a variety of particular implementation assignments can be used based on the way the discussion in class has shaped up, the level on which the class was taught, and the available computer architecture. The important point is that each student will have at least one working code for the parallel sort. Multiple version of parallel sorting, versions written using various parallelizing tools (on the same machine), versions written using the same parallelizing tool (for different machines), versions written in different programming languages can also be used in different courses. Problem description is presented in Appendix 1.

Homework Assignment:
Run the code for multiple numbers of processors and multiple numbers of data entries (details depend on the available architecture). Study the performance characteristics of the code. Compare the results to the theoretical predictions. Write a technical report describing your findings (using a format based on a professional journal).

In case of one of the other possibilities suggested in the implementation section, the general framework of the homework should remain the same: multiple runs, analysis of the results, and paper. The particular aim of the assignment will depend on the curricular needs. The important point is that it is necessary to perform the experiments on a real machine, analyze the experimental data and sharpen the technical writing skills.

5.2 Distributed Processing Examples Using PVM

In order to prepare PVM teaching examples and compare their performance to existing multiprocessors, several parallel problems were implemented in this system.

(a) Calculation of Π

The first exercise involves the approximation of Π by numerical integration. A similar exercise is presented as Module 14 in [12]. Students experiment with both the sequential and parallel cases. The sequential case computes 8000 partitions to approximate the area under the curve from 0 to 1. In the parallel case, the master spawns two processes where one does 4000 partitions from 0 to 0.5 and the other does 4000 partitions from 0.5 to 1, then the two sums are added. The master sends two messages to the slave workers and waits for two messages from them. In this exercise the students explore granularity (i.e. coarse granularity, a few messages and a lot of computation). The exercise is very simple and allows students to measure timings on different workstations and compare the results. One possible extension of such experiments is to use an ATM network of Sun workstations that we have conducted at the High Performance Computing Laboratory of Michigan State University, East Lansing.

(b) Block Matrix Multiplication

Matrix multiplication is a computation-intensive application. On shared-memory multiprocessors, rescheduling algorithms work well and exhibit good speed-up characteristics with an increase in the number of processors. On distributed memory machines, matrices are decomposed into sub-blocks and multiplied, and a regular communication pattern between the processing elements helps minimize the overheads. A detailed description and analysis of block matrix multiplication on hypercube architectures can be found in programming manuals for CM-2 and CM-5 machines. A modified version of the block matrix multiplication algorithm was implemented on the PVM system and executed on various hardware combinations.

(c) Computation of Mersenne Primes

Fr. Marin Mersenne (1588-1648) was among the mathematicians of the early 17th century who worked on the problem of perfect numbers. A Mersenne prime is a prime of the form $M(n) = (2^n - 1)$ for some prime n. Thus 3, 7, 31, 127 are the first four Mersenne primes. At the time of this writing, the largest known Mersenne prime was M_{859433} with 258716 digits, found by Paul Gage and David Slovinski, with the aid of CRAY-XMP at the Lawrence Livermore National Laboratory. The interest in finding Mersenne numbers is growing due to some potential applications of those numbers in cryptography. We developed a parallel algorithm and program based on a known Lucas-Lehmer sequential algorithm, for a primality testing of Mersenne numbers.

(d) The Burg Algorithm

The Burg algorithm is a linear signal-processing procedure for fitting an autoregressive model to a time-series data set. An autoregressive model of an order m is given by

$$x_n + a_{m1} \cdot x_{n-1} + \ldots + a_{mm} \cdot x_{n-m} = e_{mn}$$

where x_n is the autoregressive process, $a_{m1} \ldots a_{mm}$ are the process parameters, and e_{mn} is white noise. The purpose of the Burg algorithm is to estimate the a_{mi} coefficients. The key to the parallel implementation of this algorithm is the fact that the autoregressive model can be realized using direct implementation or the lattice structure. The Burg algorithm is widely used in various areas of digital signal processing (i.e. seismic, biomedical, speech etc.). Any algorithm that involves a convolution or a correlation operation will very likely have this algorithm as a major component.

6 Experience

First response from the students, though rather limited, is very positive. Most of them are excited about being able to use and develop parallel and distributed applications to meet the market demands. Students enjoy experiencing the variety of paradigms and understand their importance. They appreciate the broad base of knowledge and experience that they are able to get. The approach also gives students a better understanding of concepts taught throughout the curriculum. Topics such as asynchronous processes, performance analysis, and data structures are reinforced. Because parallel and distributed computing is a very dynamic discipline and there are still many important unresolved issues, it is an excellent basis for independent study or project work for the best students.

From the instructor's perspective, students gain a good understanding of the basic concepts of parallel/distributed computing and the various programming paradigms. They develop experience in programming that belongs to both distributed-memory and shared-memory models. The exposure to different programming paradigms and systems gives our students broader knowledge of computing and awareness of the differences in the basic approaches to parallel and distributed software development.

One specific lesson regarding the contents of the curriculum is worth mentioning. There seems to be a recently emerging tendency for the split in computing models for the high performance environments. On one hand, some of the vendors (Convex, Kendall Square Research, Cray Research) move toward the shared virtual memory model. In this model of computation, the unified address space is provided (on the software development level) to the programmer. At the same time, the physical layout of the memory is distributed. On the other hand, the heterogenous network computing model represented by the popular packages like PVM and P4, as well as the supporting visual development environment Hence, seem to steadily gain popularity. Taking these developments into consideration, it is particularly important to introduce students cohesively to both models of computation.

7 Conclusion

In this paper, we presented and illustrated our effort to structure the CS curriculum according to the major objective to produce CS graduates who think in terms of concurrency and parallelism when developing applications. We adopt a software engineering view to shape the curriculum by going top-down from applications and development methodologies to hardware architectures and interconnects. In brief, our approach to teaching parallel/distributed computing can be characterized as follows:

- Understand parallel/distributed computations
- Define your teaching objective
- Select a range of topics
- Map topics onto courses
- Select appropriate tools and teaching methods.

We discussed one particular implementation attempt in the core courses at one of the institutions (UTPB). The major obstacle in successful implementation of this approach is the lack of commercial tools suitable for education. Innovative teaching methods, such as the gradual use of programming/development examples (from demos through team projects), emphasis on recurring concepts, and multimedia techniques proved to be particularly valuable in our educational environment.

We are aware, however, that the implementation of the entire project will undergo a very lengthy and painful process. The immediate next step in continuing this work is to integrate software engineering concepts with computing needs of non-computing disciplines, such as physics and chemistry, to help educate students in a broader area of, what is called, computational science [18].

References

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

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

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

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

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

[6] Jensen K., G. Rozenberg (Eds.), High-Level Petri Nets: Theory and Application, Springer-Verlag, Berlin, 1991

[7] Kennedy K. et al., Parallel Computation: Practice, Perspectives and Potential, CRPC Short Course (7 videotapes), Center for Research in Parallel Computation, Rice University, Houston, TX, 1994

[8] Kumar V. et al. Introduction to Parallel Computing, Benjamin/Cummings, Redwood City (CA), 1994

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

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

[11] Miller R., The Status of Parallel Processing Education, Computer, Vol. 27, No. 8, pp. 40-43, August 1994

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

[13] 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

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

[15] Paprzycki M., J. Zalewski, Teaching Parallel Computing without a Separate Course, Proc. NSF Workshop on Parallel Computing for Undergraduates, pp. 19/1-18, C. Neveson (Ed.), Colgate University, Hamilton, NY, June 22-24, 1994

[16] Sunderam V.S. et al., The PVM Concurrent Computing System: Evolution, Experiences, and Trends, Parallel Computing, Vol. 20, pp. 531-545, 1994

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

[18] Umar V.M. (Ed.), Computational Science Education Project, Mosaic Universal Record Locator (URL): http://csep1.phy.ornl.gov/csep.html

[19] 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

[20] 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

[21] 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

[22] Zalewski J. (Ed.), Advanced Multimicroprocessor Bus Architectures, IEEE Computer Society Press, Los Alamitos (CA), 1994

Appendix 1. Odd-even Transposition Sort [8]

We want to sort n elements (n is even). The odd-even transposition sort is an extension of the bubble sort and needs n steps to sort the data. In the first phase elements $(a_1, a_2), (a_3, a_4), \ldots, (a_{n-1}, a_n)$ are compared and exchanged. In the second phase, the pairs $(a_2, a_3), (a_4, a_5) \ldots, (a_{n-2}, a_{n-1})$ are compare-exchanged. The process is then repeated. After n steps, the data are sorted so that the algorithm's complexity is $O(n^2)$. The pseudocode for this algorithm is as follows:

```
Procedure ODD_EVEN(n)
for i := 1 to n do
begin
    if i is odd then
        for j := 0 to n/2 - 1 do
            compare-exchange(a2j+1, a2j+2);
    if i is even then
        for j := 1 to n/2 - 1 do
        compare-exchange(a2j, a2j+1);
end
```

To parallelize this algorithm one needs to observe that the compare-exchange operation can be performed in parallel. Assume that there are n processors, and that *id* is the processor number. In the odd phase, each processor with odd label compare-exchanges elements with the processor with an *id* by one larger ("to its right"). In the even phase, each processor with an even *id* (except the nth one) compare-exchanges data with its right neighbor (processor with an *id* by one larger). The pseudo-code for this routine is as follows:

```
Procedure ODD_EVEN(n)
id := processor's label
for i := 1 to n do
    begin
        if i is odd then
            if id is odd then
                compare-exchange(with id+1);
    if i is even then
        if id is even then
            compare-exchange(with id+1);
end
```

Assuming that the compare-exchange operation is of the order $O(1)$ and since n steps of the algorithm will be performed, the parallel complexity of this algorithm is $O(n)$. In case when there are more data elements than processors the data need to be initially divided equally between processors and sorted using a fast sorting algorithm (e.g. quicksort). Then, processors perform p phases of odd-even compare-split operations.

Appendix 2. Partial List of Concurrent/Parallel/Distributed Systems (in Public Domain)

1. Multi-Pascal (Interpreter and Debugging Tool)
   Host operating system: MS-DOS
   Availability: 5.25-inch diskette attached to a book
   Documentation: "The Art of Parallel Programming" by B. Lester,
   Prentice-Hall, 1993, ISBN 0-13-045923-2
2. Pascal-FC (Functionally Concurrent)
   Host operating system: MS-DOS
   Availability: ftp.brad.ac.uk:/software/msdos/pfc-pc.zip
   Documentation: "Concurrent Programming" by A.Burns and G.Davis,
   Addison-Wesley, 1993, ISBN 0-201-54417-2
3. Modula-P (+ Parallaxis, and Petri nets simulator)
   Host operating system: Unix
   Availability: ftp.informatik.uni-stuttgart.de:/pub/modula-p
   /pub/parallaxis and /pub/petri-nets
   Documentation: "Parallel Programming" by Thomas Braunl,
   Prentice Hall, 1993, ISBN 0-13-336827-0
4. SR (Synchronizing Resources)
   Host operating system: Unix
   Availability: cs.arizona.edu:/sr
   Documentation: "The SR Programming Language" by G.R. Andrews
   and R.A. Olsson,
   Benjamin/Cummings, 1993, ISBN 0-8053-0083-X
5. Erlang (not Public Domain but freely distributed)
   Host operating system: SunOS and Solaris
   Availability: euagate.eua.ericsson.se:/pub/eua/erlang/info
   (for information how to obtain a copy)
   Documentation: "Concurrent Programming in Erlang"
   by J. Armstrong, M. Williams & R. Virding,
   Prentice Hall, 1993, ISBN 0-13-285792-8
6. P4 (Portable Programs for Parallel Processors)
   Host operating system: Unix
   Availability: info.mcs.anl.gov:/pub/p4
   Documentation: User's Guide to the P4 Programming Systems,
   by R. Butler and E. Lusk, Report ANL-92/17,
   Argonne National Laboratory, October 1992
7. PVM (Parallel Virtual Machine)
   Host operating system: Unix
   Availability: E-mail the command: send index from pvm3
   to netlib@ornl.gov to receive instructions.
   Documentation: PVM 3.0 User's Guide and Reference Manual,
   by A. Geist et al., Report ORNL/TM-12187,
   Oak Ridge National Laboratory, February 1993