

A MODERN COURSE ON PARALLEL AND DISTRIBUTED PROCESSING

Lubomir Ivanov
Department of Computer Science
Iona College
715 North Avenue, New Rochelle, NY 10801
tel. : 914-633-2342
email: livanov@iona.edu

ABSTRACT

The rapid advances in both the development and the use of parallel computing systems has led to the need for highly trained professionals with a knowledge of both the hardware and the software aspects of parallel and distributed computing. We present a course aimed at preparing students for the demands of the workplace by giving them hands-on experience working with parallel computers and software. Additionally, this course emphasizes the essential connections between Computer Science and other disciplines, such as Biology, Biochemistry, Physics and Astronomy, where the use of parallel computing is commonplace.

I. INTRODUCTION

Parallel computing, which for many years was considered a purely academic endeavor is gradually becoming a necessity of modern mainstream computing. There are many reasons for this: The current level of microprocessor technology seems to be finally reaching the limit of the clock speed at which CPUs can be operated with reasonable cooling. Classical parallel computing techniques such as SIMD have been reintroduced in the context of general-purpose microprocessors. On the other hand, the rapid advances in Biology, Chemistry, Physics, and Astronomy have dramatically increased the need for ever-more powerful computational platforms for modeling and simulating complex real world phenomena and processes. Immerging new fields, such as Biocomputing, have spanned the boundaries between Computer Science and other disciplines, demanding greater breadth and depth of knowledge and skills from the practitioners. These skills include an in-depth understanding of parallel computing both from a hardware and software point of view, and can only be acquired through extensive hands-on experience working on parallel systems and projects.

To prepare such highly trained professionals, capable of designing, implementing, and operating parallel systems at the software and hardware level, Computer Science programs must include a wider coverage of parallel computing in their curricula. Traditionally, courses on parallel computing are taught as electives to seniors or graduate students. A typical course on parallel computing usually includes a historical overview of the major architectural approaches and historically significant machines, coupled with a brief introduction to various parallel programming paradigms. In the

context of modern parallel architectures and parallel programming needs, such a treatment of the subject is insufficient. While learning about the ILLIAC IV and the Cray YMP may be interesting from the point of view of studying the diversity of design decisions employed in these early architectures, it does little to prepare students for the demands of modern parallel processing.

In this paper, we outline the structure and contents of an upper-level undergraduate course on Parallel and Distributed Computing, which places the emphasis primarily on modern techniques for parallel and thread programming, backed up by a sufficient exposure to modern parallel architectures and theory. The course centers around several relatively large-scale projects, based on problems of current interest to the scientific community. These projects are to be implemented by students using languages that support threads (e.g. Java) and parallel programming libraries such as the Message Passing Interface (MPI). In addition to developing parallel algorithmic thinking and programming skills, students learn to estimate the expected performance of their software by analyzing the running time of their algorithms as a function of the number and type of processes, physical CPUs, and the communication capability of the interconnection network. Students become familiar with a number of key modern parallel architectures, study certain aspects of distributed operating systems development, and understand the issues of mapping virtual processes to the underlying physical hardware of the machine in order to maximize performance.

In the remainder of this paper, we outline the need for parallel and distributed computing, discuss a basic classification of parallel architectures and software, consider some modern trends in parallel computing, and present the specifics of our Parallel and Distributed Computing course.

II. THE NEED FOR A PARALLEL AND DISTRIBUTED COMPUTING COURSE

The current level of microprocessor implementation technology has reached sub-100nm range. The ever-increasing circuit complexity, packed into an ever-smaller space, invariably leads to problems such as quantum mechanical effects and heat, that, at the current level of technology, are difficult or impossible to overcome. While companies are actively researching alternate implementation technologies and materials (e.g. “quantum well” transistors based on InSb [1]), the current microprocessor implementation technology seems to have reached the peak clock speed at which conventional cooling can still be employed. In fact, that is precisely the reason why the latest Intel Pentium-D chips run at a lower (up to 3.4GHz) clock speed compared to their predecessors (max 3.8GHz). They do, however, offer dual core technology which, when properly used by the software, promises to significantly improve performance. Other traditional parallel processing concepts have found their way into the instruction sets of modern CPUs such as the MMX and SSE (Streaming SIMD Extensions) in the Intel and AMD processors’ instructions sets. Non-traditional, highly parallel architectures, such as Neural Networks and Cellular Automata have helped deal with problems for which algorithms are either difficult or impossible to design. At the software level, threads – the ability to split a process into independent and concurrent tasks – have been widely accepted, and thread support has been incorporated into most powerful programming languages such as C++ and Java. The majority of modern general-purpose as well as embedded applications are multithreaded, which contributes greatly to their efficiency. On the other hand, parallel processing libraries such as PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) have become the backbone for developing highly complex

applications for dealing with a variety of computationally demanding problems from Astronomy, Physics, Chemistry, Biology, and Economics. The recent advances in Bio-Chemistry, DNA-analysis, atmospheric research, etc. are due primarily to the availability of high-performance parallel architectures and supporting software.

In general, with traditional microprocessor technology facing fundamental limitations, and with novel computing approaches (quantum computing, etc.) still very much in an early research phase, parallel processing offers the only real opportunity for continued increase of the performance of computing systems. The computer industry – both software and hardware – has embraced parallel processing, and it is no longer considered a purely academic endeavor. Ironically, it is in the academic circles that parallel computing is still regarded with skepticism. Most engineering-type universities offer senior level electives on Parallel Processing, which, more often than not, provide a broad historical overview of the field without emphasis on the modern aspects of parallel processing. Some such courses take the “hardware” approach to parallel computing and trace the evolution of parallel architectures, from its origins in the 1960’s through the recent past, by examining a number of machines representing typical architectural approaches, e.g. the ILLIAC IV as the representative of SIMD machines, the CRAY YPM as a typical vector processor architecture, the CRAY T3D as a classic shared-memory MIMD design, etc. Small snippets of code are used to illustrate specific issues in programming the machines in question. Other Parallel Processing courses ignore the architectural aspects of parallel processing completely, and concentrate instead on exploring a number of different programming paradigms.

At most liberal-arts colleges, courses on Parallel Processing are usually not offered. The only aspects of parallel computing that students are exposed to are pipelining and superscalar processors, which are briefly covered in most Computer Organization and Architecture courses. The concept of a thread is usually mentioned in the Operating Systems course, but few courses offer students hands-on experience working with threads.

III. PARALLEL AND DISTRIBUTED COMPUTING

In this paper we take a look at the current state of parallel and distributed computing, and outline a course, which strives to strike a balance among the theoretical, hardware, and software aspects of modern parallel and distributed computing. We believe that it is impossible to cover in a single course the multitude of architectural and software approaches as well as the extensive theory that has been developed over the past few decades. Thus, we sacrifice some of the breadth of the subject for a more detailed treatment of specific areas, which are essential in present-day parallel computing.

1. The Current State of Parallel and Distributed Computing

Before we consider the contents of the course, we need to take a brief look at the state of modern parallel and distributed computing. Modern parallel architectures can be loosely classified as data-parallel or code-parallel. Data parallel architectures are those, which apply the same operation to multiple streams of data concurrently. The two main categories of data-parallel machines are vector and SIMD (Single Instruction/Multiple Data Stream) architectures. In addition, some microprocessors, such as the Intel Pentium and the AMD Athlon, provide the so called Streaming SIMD Extensions (SSE) of their respective instruction sets, which allow simultaneous operation on several integers or floating point numbers. Some non-traditional models of computation, such as Artificial Neural Networks and Cellular Automata also belong to this class.

The code-parallel architectures are those, which execute code in parallel. Depending on the grain of parallelism, these architectures fall into the following groups:

- *Instruction-level parallel architectures*: these execute multiple instructions concurrently and/or in an overlapped fashion. This class includes the pipelined, superscalar and VLIW architectures.
- *Thread/Process-level parallel architectures*: these execute multiple processes/threads concurrently, and include the so-called distributed- and shared-memory MIMD (Multiple Instruction/Multiple Data Stream) machines.

Most modern architectures often span the boundaries of these categories and offer combinations of approaches to parallelism. For example, the Intel Pentium processors are essentially instruction-level parallel architectures that employ both pipelining and superscalar processing, but also allow streaming SIMD operations for specific multimedia types of data, as well as dual-core capability, which allows a truly parallel thread execution to be scheduled by the operating system.

In recent years, a number of significant hardware and software trends have emerged and established themselves as the predominant way to employ parallel and distributed processing:

- *Cluster computing*: This approach belongs to the thread/process-level parallel architecture category (usually distributed memory MIMD). Originally known as “poor man’s supercomputing”, the idea is to use a relatively large collection of inexpensive off-the-shelf microprocessors such as Intel Pentiums or DEC Alphas and interconnect them using regular Ethernet. The performance gain is not as significant as with a true supercomputer, which uses specialized processors and optimized interconnection networks and routing algorithms, but the cost of a cluster is many times lower than that of a typical supercomputer. In recent years, cluster computing has become the predominant way of implementing high-performance systems. Most computer manufacturers such as Dell, HP, IBM, and Intel now offer relatively inexpensive clusters ranging in price from \$20,000 to about \$100,000. Taken to a higher level, the idea of cluster computing has become the basis for the design and implementation of many of today’s most powerful machines, such as Blue Gene/L[2], ASC Purple [3], and ASCI Red [4].
- *The Message Passing Interface (MPI)*[5]: Message passing is a powerful method for processes/threads to exchange data when collaborating on solving a large problem. MPI is a library specification for message passing, which has established itself as the de-facto standard in the parallel processing (in particular MIMD) community, pushing aside all the older message passing methods such as PVM. The MPI specification has been implemented into C, C++, and Fortran libraries. Java is not directly supported, but the recently developed *mpiJava* [6] interface provides the ability to interface Java programs to the standard MPI. The use of MPI allows the creation of extremely complex and powerful parallel programs. MPI is universally used when writing large-scale computing simulations of biological processes such as protein-folding or astrophysical processes such as N-body interactions. Other applications that rely heavily on the use of MPI are weather-prediction software, fluid-dynamical computation packages, and even seismic research tools.
- *Threads*: The ability to split a process into a number of lightweight, independent and concurrent sub-processes – threads – has revolutionized software development

on all platforms. Developing multithreaded applications requires a new set of skills and a new type of algorithmic thinking. On the one hand, threads offer the programmers the convenience of concentrating on implementing a particular task without worrying about how to integrate it in the larger context of an application – if two tasks need to be performed concurrently, then each is designed as a separate thread and started independently of the other. On the other hand, thread programming is not without its own set of challenges. Threads often require synchronization: Multiple threads must not be allowed to simultaneously update an object. Alternately, one thread may compute a result needed by another thread, and the interaction between the two must be carefully orchestrated. Real-time applications of threads in embedded systems raise a number of additional challenges such as timing constraints, deadlock avoidance, and general software quality control to name a few. New standards and language subsets, such as the Java 2: Micro Edition, are specifically designed to serve the need for thread processing in embedded systems.

- *High-Performance Fortran*: While code-parallel architectures rely primarily on thread processing and message passing, data-parallel architectures require a different set of standards and languages, which include convenient primitives for distributing data (vectors, matrices, etc.) among processors so that it can be manipulated in parallel. The High Performance Fortran Forum [7] has developed a number of extensions of Fortran 90, known collectively as High Performance Fortran (HPF), to help programmers write data-parallel programs. Some complex fluid-dynamic and astrophysical simulations have been implemented using HPF, but considering the overall predominance of MIMD architectures and message passing in modern supercomputing, the significance of HPF is somewhat reduced.

In view of these trends, our Parallel and Distributed Computing course concentrates on cluster computing, thread processing, and message passing. We aim to give our students a fundamental understanding of the basic concepts and challenges in these areas, and provide hands-on experience by engaging the students in projects that use the aforementioned languages, libraries, and techniques.

2. The Parallel and Distributed Computing Course

The course incorporates a lecture-, a lab-, and a project component. The lectures cover the basic theory of parallel computing as well as details of specific architectures, design decisions and tradeoffs, software paradigms, etc. The labs allow students to gain hands-on experience with the material presented during lectures by applying it in the context of small case-studies and programming assignments. The course also includes several larger projects, which students have to complete independently or in groups.

2.1 The Lecture Component

The first few lectures concentrate on performance issues: estimating execution time speedup of a parallel program compared to a serial implementation, using Amdahl's law for parallel processing, exploring the parallel complexity classes, studying the effects of communication and IO on the performance of a parallel program, etc. The idea is to give students the tools for evaluating program performance as early as possible, and to get them used to considering factors such as the number of processors, the IO bottleneck, etc. when deciding how to best parallelize their programs.

Next, Flynn's taxonomy is introduced, and the main categories of parallel and distributed architectures are discussed. Here the emphasis is mostly on understanding the conceptual differences among the various types of architectures. At this point, the notions of vector processing and message passing are introduced and illustrated with a few simple HPF and MPI programs, whose execution is traced in detail. The role of the compiler and the operating system in handling parallelism is also discussed. This lead naturally to the discussion of threads.

Threads are defined as individual sequential control flows within an executing program. User-level vs. kernel level threads are studied, and the advantages and disadvantages of each are discussed from the point of view of context switching overhead, cooperative multitasking issues, etc. Thread synchronization is studied in the context of the classical producer-consumer problem: A handshaking protocol for data exchange between a producer and a consumer process and possible problems that may arise are examined in detail and illustrated with specific code examples. Other issues such as preventing simultaneous write access to a shared resource, and thread deadlock avoidance are considered. The uses of threads in web-based applications as well as in various embedded systems are studied.

Next, the discussion centers on cluster computing. The architectures and interconnection network patterns of a typical cluster are considered, and various design decisions, potential problems and bottlenecks are discussed. A specific MPI program is used to demonstrate the extent of the effect of IO and inter-process communication on the speed of program execution. This, on the one hand, relates the theoretical material discussed in the beginning of the course to a very practical, real-world example, and, at the same time, leads to a discussion of optimal network structures for parallel processing. Some real cluster architectures are examined as well.

The discussion of cluster computing introduces students to the Message Passing Interface (MPI). A brief comparison with the Parallel Virtual Machine (PVM) library is made, and then the various aspects of MPI are introduced in the context of specific scientific problems. In particular, the classical N-body problem is presented, and various solutions are considered and compared based on the efficiency of their parallel implementation. If time permits, the instructor may choose to present other topics of current scientific interest, such as the protein-folding problem in biochemistry, and discuss strategies for designing parallel simulation algorithms – from a mathematical specification to a parallel implementation and debugging.

The last part of the course deals with non-traditional, massively-parallel models of computation. Artificial Neural Networks are compared and contrasted with Cellular Automata. Depending on the available time, a number of different Neural Network models can be discussed such as Perceptrons, Backpropagation Networks, Hopfield Networks, etc. A general theoretical discussion of Cellular Automata can lead to a project on implementing Conway's Game-of-Life 2-dimensional Cellular Automaton.

2.2 The Lab Component

The lab component is synchronized with the lecture material, but concentrates more on hands-on activities and assignments. The first lab gives students an opportunity to evaluate the performance of several simple parallel programs. The next few labs concentrate on threads, and offer students a set of increasingly more challenging assignments implementing, synchronizing, pausing and restarting threads, etc. In addition to learning about threads, the students' knowledge of Java and the Swing API

is reinforced as the complexity of the later assignments becomes significant, and knowledge acquired in previous programming courses becomes essential.

From just before the middle of the semester, students are formally introduced to MPI. Various aspects of MPI – point-to-point vs. broadcast messaging, grouping data for communication, derived types, advanced IO and file IO issues – are covered in successive labs by requiring students to complete, during the course of the lab, portions of instructor-prepared MPI programs or to write complete programs independently. Lab assignments, which cannot be completed during the allocated lab period, are expected by the next lab period.

2.3 The Projects

The projects are an essential part of the Parallel and Distributed Computing course. They not only provide students with additional hands-on experience, beyond that acquired in the lab, but allow them to see the “big picture” – the application parallel computing to large-scale, computationally intensive problems, for which sequential solutions are too time-consuming. Additionally, the connections to other areas of research, such as Biology, Chemistry, and Physics, are emphasized, and the role of Computer Science in the overall context of Science is highlighted.

Typically, three to four projects are given during the course. The first project is a multithreaded implementation of the handshaking communication protocol discussed earlier. The protocol specifies a set of rules for data exchange between a sender and a receiver through a shared buffer. The buffer capacity may be a single byte or a circular queue. In the case of a single-byte data buffer, the status of the buffer is given by a data flag: 0 for buffer empty, 1 for buffer full. In the case of the buffer implemented as a circular queue, a shared “count” variable is used to monitor the actual number of items in the buffer. The sender and the receiver are implemented as separate threads, and their proper synchronization is emphasized.

The second project is an MPI implementation of host/guest algorithm for solving the N-body problem from Physics, Astronomy and Biochemistry. The N-body problem describes the interaction of a set of N objects (planets, particles, atoms or molecules, etc.) through the forces they exert on each other. The sum of these forces acts on each of the objects by changing the object’s acceleration and therefore its spatial position relative to other objects. As a result of that change, the strength of the interaction forces is changed, etc. Solving the N-body problem for large collections of objects is an extremely computationally-intensive task, and many different algorithms have been designed to deal with the problem. The host/guest algorithm is a relatively simple (but not particularly efficient) algorithm, which is discussed during lecture and assigned to students as a MPI-based project.

The third project is an MPI-based implementation of Conway’s Game-of-Life 2D Cellular Automaton. This project is particularly fun for students, and many of them choose to implement a graphical interface, which allows them to display the evolution of the CA colonies under iteration.

3. Course-Related Challenges and Obstacles

In the context of the present-day need for parallel and distributed computing, the Parallel and Distributed Computing course is vital to the development of students as knowledgeable and skilled designers and users of parallel and distributed software and

hardware. However, implementing the course and integrating it within the curriculum does present a set of challenges and obstacles.

First, a parallel computing course implies the availability of a parallel system, on which the material can be presented and students be allowed to experiment. As indicated earlier, many hardware manufactures and vendors do offer reasonable priced clusters along with deep discounts for educational institutions. In addition, some companies and organizations offer equipment grants to qualified schools provided the need for parallel platforms can be clearly demonstrated both at the research and the teaching level. The National Science Foundation (NSF) also solicits grants for the use of the equipment available at many National Supercomputing Centers. Even if specialized cluster equipment is not available, MPI can be installed on any network of workstations/PCs and operating system (UNIX/Linux/Windows). Even on stand-alone machines, MPI can simulate message passing between virtual processes, although, from a computational efficiency point of view, this is not beneficial.

The second challenge is the students' background and prior knowledge and skills. This course requires a solid mastery of at least two major programming languages (Java and C/C++), a firm understanding of computer architecture, a good working knowledge of discrete mathematics and some aspects of calculus, and at least some background in areas of science such as Physics, Chemistry and Biology. Even though this may seem like a lot of prerequisites, most students, by the time of their senior year, have a mastery of at least one programming language, have taken courses on Discrete Math, Calculus, and Physics, as well as a course on Computer Architecture. Thus, the Parallel and Distributed Computing course is not beyond the reach of most Computer Science seniors, and will, in fact, serve to integrate the material they have covered in all the other courses into a coherent body of knowledge.

There are, however a number of non-Computer Science majors, who may have significant interest in the course, as it provides them with the skills for working on complex problems in other areas of Science. Typically, these students have the necessary science and math prerequisites, but lack solid programming skills and computer architecture knowledge. The best way to serve these students is to offer a concentration in Parallel Computing and Simulation, such that students are required to take two or three programming courses (e.g. CS1, CS2, Data Structures and Algorithms) before taking the Parallel and Distributed Computing course.

The next challenge is the amount of preparation the instructor of the course faces. The course spans the boundaries of several Computer Science sub-disciplines, such as Computer Architecture, Programming Languages, Operating Systems, etc, and presents ideas from other areas of science, such as Chemistry, Physics, and Biology. The instructor must be well-versed in all these areas in order to provide students with a truly useful experience. Some of the books and articles helpful to the instructor are listed in the references section of this paper [8-16].

Integrating the course into the curriculum also presents a challenge. At this time, few departments will agree to offer a required course on parallel processing. On the other hand, the number of electives, particularly in liberal arts colleges, is usually very small. As described, the Parallel and Distributed Computing course merits 4 credits (3 lecture and 1 lab). A four credit elective may not be considered acceptable, particularly by the administration, since it contributes significantly to the teaching load of the instructor, that may otherwise be required to teach an extra course. The answer

to these challenges is to emphasize the complexity, breadth, and depth of the subject matter, and its utmost significance in the context of preparing students for the demands of the modern day workplace.

IV. CONCLUSIONS

In this paper we outlined the development of a course on parallel and distributed computing as a senior elective with lecture, lab, and project components. The course stems from the ever-increasing need for computer scientists, engineers, and practitioners well-versed in the intricacies of modern parallel processing, who can integrate themselves and assist in computationally-intensive projects of national and international significance such as DNA analysis, biomedical applications development, and even space exploration. The course presents challenges both to the instructor and the students, but is an overall valuable and rewarding experience for all.

REFERENCES

- [1] T. Ashley, et al, “*Novel InSb based Quantum Well Transistors for Ultra-High-Speed, Low Power Logic Applications*”, Int. Conf. on Solid-State and IC Technology, Beijing, 2004
- [2] <http://www.research.ibm.com/bluegene/index.html>
- [3] <http://www.llnl.gov/asci/platforms/purple/>
- [4] <http://www.sandia.gov/ASCI/Red/>
- [5] <http://www-unix.mcs.anl.gov/mpi/>
- [6] <http://aspen.ucs.indiana.edu/pss/HPJava/mpiJava.html>
- [7] <http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm>
- [8] D.Culler, J.Singh, A.Gupta, “*Parallel Computer Architecture : A Hardware/Software Approach*”, Morgan Kaufmann
- [9] G.Almasi, A. Gottlieb, “*Highly Parallel Computing*”, 2nd edition, Addison Wesley Longman
- [10] D.Sima, T.Fountain, P.Kacsuk, “*Advanced Computer Architecture: A Design Space Approach*”, Addison-Wesley
- [11] P.Pacheco, “*Parallel Programming with MPP*”, Morgan Kaufmann
- [12] W.Gropp, E.Lusk, A.Skjellum, “*Using MPI: Portable Parallel Programming with the Message Passing Interface*”, 2nd edition, MIT Press
- [13] G.Karniadakis, R.Kirby II, “*Parallel Scientific Computing in C++ and MPI : A Seamless Approach to Parallel Algorithms and their Implementation*”, Cambridge University Press
- [14] S.Oaks, H.Wong, “*Java Threads*”, 3rd Edition, O'Reilly Media, Inc.
- [15] V.Garg, “*Concurrent and Distributed Computing in Java*”, Wiley-IEEE Press
- [16] A.Wellings, “*Concurrent and Real-Time Programming in Java*”, Wiley Publishing