ED 386 165 IR 017 293

AUTHOR	Sheary, Kathryn Anne
TITLE	Parallel Processing at the High School Level.
PUB DATE	95
NOTE	88p.; Master's Research Paper, University of Illinois at Springfield.
PUB TYPE	Dissertations/Theses - Undetermined (040)
EDRS PRICE	MF01/PC04 Plus Postage.
DESCRIPTORS	Computer Literacy; *Computer Science Education; Computer Software; Computer Uses in Education; High Schools; *High School Students; *Programming; Surveys; Teacher Attitudes
IDENTIFIERS	Illinois; *Parallel Processing

ABSTRACT

This study investigated the ability of high school students to cognitively understand and implement parallel processing. Data indicates that most parallel processing is being taught at the university level. Instructional modules on C, Linux, and the parallel processing language, P4, were designed to show that high school students are highly capable of not only understanding, but of applying parallel processing. Eight high school students, 11th and 12th graders, were study participants. Students studied, evaluated, and executed programs in C and P4 on the Linux operating system. A complete description of the six instructional modules is provided. Overall, the results of the instructional modules clearly indicate that high school students can understand and implement parallel processing. An unanticipated positive result of this project was the high level of student motivation that resulted. Students pursued additional avenues beyond the scope of this project and spent many hours outside of class on the project. Also included in the study was a survey of 30 high schools in Illinois concerning teaching parallel processing and other programming languages. Results shows that most high school teachers are unaware of the existence of parallel processing and/or reluctant to advance their own knowledge in this area. Appendices include the instructional modules, introductory activities, performance tests, surveys, and C, Linux/Unix, and P4 handouts. Four figures and three tables illustrate data. (Contains 16 references.) (MAS)

PARALLEL PROCESSING AT THE HIGH SCHOOL LEVEL

Presented in Partial Fulfillment of the Requirements
for the Degree Master of Arts of Mathematical Sciences
with a concentration in Computer Science

By

Kathryn Anne Sheary

University of Illinois at Springfield

1995

I would like to express my thanks to Dr. Mims and Dr. Miller for their support during my graduate project and throughout my masters work.

A very special thanks to the following high school students for their cooperation and help with this project:
Jolyn Braden, Rhonda Braden, Steve Brown, Tim Brown, Lisa Hoffman, Heather Huber, Jon Mullins, and Chris Reed.

An extra special thank you is extended to my husband and children for their support, love, and patience during the long process of obtaining my masters degree.

CONTENT

ILLUSTRATIONS

ILLUSTRATIONS CONT.

ABSTRACT

This study investigated the ability of high school students to cognitively understand and implement parallel processing. Data indicates that most parallel processing is being taught at the university level. No literature could be found to indicate that parallel processing is being taught on the high school level. The results show that most high school teachers are unaware of its existence and/or reluctant to advance their own knowledge in this area. Instructional modules were designed to show that high school students are highly capable of not only understanding, but also applying parallel processing. An unanticipated positive result of this project was the high level of student motivation that resulted. Students pursued additional avenues beyond the scope of the project and spent many hours outside of class on the project. This study reports the results of teaching parallel processing to high school students.

Introduction

Can high school students understand and implement parallel processing? Public high school education does not emphasize computer programming in the curriculum. This project proved that despite this trend, high school students can succeed not only at programming but at parallel processing.

The objective of this project was to demonstrate that high school students can understand and implement parallel processing. The task of demonstrating this objective was accomplished through the use of small teaching modules and lessons on C, Linux, and the parallel processing language, P4. Students studied, evaluated, and executed programs in C and P4 on the Linux operating system. Performance tests conveyed that the students successfully met the objective.

It was worthwhile to demonstrate that high school students could do more than introductory programming in Basic or QBasic. Since students at the high school level are capable of doing much more, then principals and superintendents of public school districts may help to develop this aspect of the computer science curriculum. Administrators have to be convinced of the importance of computer programming at the high school level.

Rationale and Significance

The total elimination of programming at the high school level is a possibility. First of all, technology is changing faster than most schools can keep up with financially. School administrators tend to "jump" on popularity band wagons to improve their status rating with neighboring school districts. Some might say they flow with the "technological trend" of the time. Two years ago (1993-94) the big trend was to teach multimedia classes at the high school level. This past year (1994-95) the big question was "when can we get Internet for our students?" Now the Maroa-Forsyth district, in which the author teaches, is going to purchase new computers to access the Internet. Suggestions to start a new telecommunications class that would include how to use the Internet have been vetoed until some future date "down the road." This action indicates that administrators who are not educated in computer science tend to "put the cart before the horse" just to keep up with the popular trend. High school administrators who do have an understanding of technology develop Technology Plans before they go out and buy computer hardware and software! Ten years ago when the district started offering computer science we had more sections of programming classes than computer application classes. In the Maroa-Forsyth school district there have been cutbacks in the computer programming curriculum over the last three years. During the 1994-95 school year, these offerings were reduced to one section of programming. This possible elimination of programming seems to indicate a need to prove to administrators that high school students need to take courses that introduce them to computer science. Programming success at the high school level can lead to further study at the university level which can result in quality programmers being employed in the workplace and helping to further advance technology. Data is not available from the surveys of past graduates. However, several former graduates have completed college degrees in computer science and are employed in computing related careers. To continue to prepare students for careers in computer science there is a need for high school programming courses that introduce the students to innovations in the field.

Problem Statement

Parallel processing is an exciting area of computing that is available and can be utilized to stimulate the interest of high school students to pursue careers in the field of computer science. Can high school students master the concepts of parallel processing? This problem requires students to understand the C Programming language, the Unix/Linux operating system, and P4, for parallel processing.

Literature Review

A review of the related literature seems to indicate this is a first attempt to formally introduce parallel processing at the high school level. However, according to IEEE Computer in August, 1994, an organization was formed in 1992 known as The IEEE Technical Committee on Parallel Processing (TCPP). It was formed so scientists and engineers could exchange ideas on parallel processing. One of its objectives was to establish guidelines for parallel processing in education at the post-secondary level. The TCPP published a report that contained course descriptions and class listings from more than 70 sites including undergraduate institutions, university centers, research institutes, and corporations that offer educational opportunities in parallel processing. The report is included in table 1. After collecting this information, the TCPP found the following:

♦ "At four-year colleges, courses in parallelism are beginning to appear. Many of these colleges have access to either a small parallel machine (transputer-based) or a small network of workstations. These machines are used for projects in the courses and for independent studies on parallelism.

- The trend at university centers in terms of graduate-level education appears to provide an array of courses, including parallel algorithms, parallel programming, and parallel architectures, to name a few. The students have access to machines either at the university or at a National Science Foundation-sponsored site. Typical machines used in the classroom include those manufactured by Maspar, Thinking Machines (both the CM-2 and CM-5), Intel, and nCube.
- The trend at university centers in terms of undergraduate education appears to include aspects of parallelism in a variety of traditional courses (for example, algorithms, architecture, operating systems, and programming languages) and to provide at least one introductory course in parallel processing" (Miller 1994, 40).

TABLE 1

This table serves to briefly categorize some of the information found in the full TCPP report.

● = Subjects covered in undergraduate courses
⧓ = Subjects covered in graduate courses
NS = Number of courses unspecified

	Undergraduate courses	Graduate courses	Compilation techniques	Dataflow computers	Distributed algorithms	Distributed/shared memory architectures	Parallel algorithms	Parallel architectures	Parallel operating systems	Parallel programming	Performance evaluation	Scientific computing (HPC)	Software utilities
Auburn University	3	4				● ⧓	● ⧓	● ⧓		● ⧓			
Bucknell University	1			●		●	●	●		●			
California Institute of Technology	NS	NS			⧓		⧓			⧓			
Clarke College	1					●	●	●		●	●		
Colgate University	1					●	●	●					
Colorado State University	1	2				● ⧓	● ⧓	● ⧓		● ⧓			
Connecticut College	1					●	●	●					
Cornell University	1	3	⧓			⧓	⧓	⧓				● ⧓	
Dartmouth College	NS	1				⧓	⧓	⧓					
Drexel University	1									●	●		
Eastern Connecticut State Univ.	2				●	●	●	●					
Edinburgh Parallel Computing Cntr.	1					●	●	●		●			●
Gallaudet University	1					●	●	●				●	
Gettysburg College	1					●	●	●					
Harvard University	1					●	●	●				●	
Illinois State University	1					●	●	●					
Iowa State University	1	5			⧓	● ⧓	⧓	● ⧓	⧓	⧓	● ⧓	●	
Macalaster College	1					●	●	●		●			
Michigan State University	2	6			⧓	● ⧓	⧓	● ⧓	⧓	●		● ⧓	●
Michigan Technology University	1					●	●	●			●		●
Mississippi State University		5				⧓	⧓	⧓	⧓	⧓			
New Jersey Institute of Technology		5				⧓	⧓	⧓	⧓				
North Dakota State University	1		●			●	●	●		●			

TABLE 1 CONT.

● = Subjects covered in undergraduate courses
■ = Subjects covered in graduate courses
NS = Number of courses unspecified

	Undergraduate courses	Graduate courses	Compilation techniques	Dataflow computers	Distributed algorithms	Distributed/shared memory architectures	Parallel algorithms	Parallel architectures	Parallel operating systems	Parallel programming	Performance evaluation	Scientific computing (HPC)	Software utiliti
Northeastern University	1	1				● ■	● ■	● ■					
Northwestern University	NS	NS				● ■	● ■	● ■	●			■	
Ohio University, Athens	NS	1				● ■	● ■	● ■					
Oregon Graduate Inst. Sci. & Tech.		3	■			■		■		■		■	
Oregon State University	1	5				● ■	● ■	● ■		● ■			
Pennsylvania State University		3				■	■	■					
Polytechnic University, Brooklyn		1				■	■	■					
Purdue University	2	8	■		■	● ■	● ■	● ■		● ■			■
Rice University	2	5	■		■	● ■	●	● ■	●			● ■	
Rochester Institute of Technology	2					●	●	●		●			
St. Bonaventure University	1					●	●	●					
St. John Fisher College	1			●		●	●	●					
San Jose State University	1	1				● ■	● ■	● ■		● ■			
Sangamon State University	1		●			●	●	●	●				
SUNY-Buffalo		5	■			■	■	■	■	■			
SUNY-Geneseo	1					●	●	●					
SUNY-Oswego	1				●	●	●	●		●			
Texas Institute of Technology		2				■	■	■		■			
University of Arizona	1	1								■			
University of Arkansas		2				■		■		■			
University of Bergen, Norway	1					●	●	●					
UC Berkeley	NS	3				■	■	■				■	
UC Irvine		2				■	■	■		■			

TABLE 1 CONT.

● = Subjects covered in undergraduate courses
■ = Subjects covered in graduate courses
NS = Number of courses unspecified

	Undergraduate courses	Graduate courses	Compilation techniques	Dataflow computers	Distributed algorithms	Distributed/shared memory architectures	Parallel algorithms	Parallel architectures	Parallel operating systems	Parallel programming	Performance evaluation	Scientific computing (HPC)	Software utilities
UC Los Angeles	2	3	●			●■	■	●■		●■			
UC San Diego	1	4				●■	●■	●■		■		■	
UC Santa Cruz	NS	1			■	■		■		■			
University of Central Florida	6	3				●■	●■	●■		●		●	
University of Colorado	2					●	●	●				●	
University of Delaware	2	3	■			●■	■	●■		●■	●■		
University of Edinburgh		1				■	■	■		■			
University of Florida		2				■	■	■					
University of Houston, Downtown	1					●	●	●					
University of Idaho	1	1				●■	●■	●■		●■			
Univ. of Illinois, Urbana-Champaign	4				●	●	●	●		●		●	
University of Iowa	NS	1								■			
University of Minnesota	2	1				●■	●■	●■				●■	
University of Nebraska, Omaha	2					●	●	●					
University of Notre Dame		2				■	■	■		■			
University of Oklahoma	1	2				■	■	■				●	
University of San Francisco	1				●	●		●				●	
University of Southern California		5		■		■	■	■		■			
University of Tennessee	6					●	●	●		●		●	●
University of Wisconsin, Madison		2				■	■	■		■		■	
Union College	1	1				■		■		●■			
Walla Walla College	1					●	●	●		●			
Wellesley College	1					●	●	●				●	

Based on the written documentation related to the college level, a portion of this project focuses on documenting an attempt to introduce parallel programming at the high school level. A survey instrument was developed to collect data from several schools in central Illinois. The results of the survey indicated the current level of computer programming and whether there is instruction in parallel processing taking place in the area schools. [See Appendix G, p 78].

First we will attempt to answer the question "What is parallel processing?" "Parallel processing is a computer processing technique in which many operations are performed simultaneously so the overall computation time is reduced" (Gehringer 1994). In serial processing, computation is performed in sequential order. Parallel processors have many interconnected processors. A coarse-grained parallel processor contains a small number of powerful processors. A fine-grained or massively parallel machine contains thousands of microprocessors.

Parallel processing was first hypothesized in the late 1940's and early 1950's. Parallelism that today would be called horizontal microcode "...appeared in Turing's 1946 design of the Pilot ACE" (Ramakrishna 1993, 12). It was described by Wilkes and Stringer in 1953 when they wrote, "In some cases it may be possible for two or more micro-operations to take place at the same time" (Ramakrishna 1993, 12). Researchers then believed they could build a computer to simulate the interconnected neurons of the human brain. Electronic models of neural networks were built modeled on a small number of neurons with limited connections. Early first-generation computers had architecture that used bit-serial main memory and bit-serial central processing units (CPU). Each bit was read from memory and arithmetic was performed on a bit by bit basis. Bit-parallel memory was developed which led to bit-parallel arithmetic. Magnetic core memory was developed as bit-parallel memory and used commercially by IBM in 1955. Input/Output instructions were executed by the CPU on these computers. They had one arithmetic logic unit in the CPU and could perform only one function at a time. (Ramakrishna 1993, 12)

Transistorized computers were developed in the 1960's. This led to successful commercial machines that used the available hardware to provide instruction level parallelism at the machine language level. In 1963 Control Data Corporation developed its CDC 6600 which had ten functional units. Any one of these could execute in a certain cycle even if the others were still processing data. This was the supercomputer of its time. In 1965 the U.S. Department of Defense needed a computer that could simulate nuclear devices because there was a ban on nuclear testing in 1962. They needed a computer that could deliver about 100 MFLOPS (million floating-point operations per second), so the Star-100 was developed. General Motors was also interested in two Star-100's if they could be done by 1971. They did not make the deadline. The Star-100 became fully operational in 1973. In the late 1960's IBM introduced and delivered its IBM 360/91 machine. It offered less instruction-level parallelism than the CDC 6600 and was not as commercially successful. An important development in computer architecture was the use of an independent I/O processor to handle I/O operations. The CPU sent I/O instructions to the I/O processor which worked independently, freeing the CPU for other computational tasks. The next architectural advance was the development of interleaved and cache memory. A memory unit is divided into a number of modules in interleaved memory. Each module has its own addressing circuitry and can be accessed simultaneously. The cache memory is a small, fast memory section placed between the CPU and main memory. The cache memory reduces the time the CPU must wait for data to arrive from memory. Data accessed most frequently is kept in cache memory. Computer speed was increased in these later generation computers by using multiple functional units through instruction and data pipelining. The functional units are independent and operate simultaneously. "Instruction pipelining allows more than one instruction to be in some stage of execution at the same time" (Chaudhuri 1992, 4). Each instruction execution can be divided into various phases such as fetch instruction, decode instruction, operand fetch and arithmetic logic execution. In a pipelined computer successive instructions are executed in an overlapped manner. (Chaudhuri 1992, 5)

In the early 1970's the CDC 7600 designed by Seymour Cray and IBM's 360/195 were the frontrunners in supercomputers. The CDC 7600 did some parallelism, but was mostly a serial computer. In 1976 the Cray-1, built by Cray Research Inc., took over large computation from the CDC 7600. Vector capabilities were added to the Cray-1. A vector computer allows operations on both vectors and scalars. Vectors are streamlined from memory into the CPU and pipelined arithmetic units perform the operation. Other pipelined vector computers were the Cyber-205 and Fujitsu-200. Texas Instruments Advanced Scientific computer was a vector machine also created in the early 1970's. It used a pipeline architecture. The ILLIAC IV designed by the University of Illinois used distributed array processors. It had four quadrants each having 64 processors.

In 1973 one of these quadrants was built and installed a NASA Ames Research Center. The term instruction level parallelism was used more frequently for multiple processor parallelism and for regular array and vector parallelism in the mid 1970's. The Parallel Element Processing Ensemble Supercomputer was built by Burroughs for the Ballistics Defense Technology Center in 1976. In 1977 and 1978 Burroughs tried to get in the supercomputer market by building the Burroughs Scientific Processor. International Computers Limited developed around this same time the DAP (Distributed Array Processor). All of the array processors discussed in the 1970's were SIMD (Single Instruction Multiple Data) machines. (Lazou 1986, 30)

At this point it should be clarified that computer systems can be identified in four distinct classes. They are (1) <u>SISD</u> - Single Instruction Single Data, these computers have one processor and one ALU; (2) <u>SIMD</u> - Single Instruction Multiple Data, this includes computers with a single program control unit and multiple ALU's; (3) <u>MISD</u> - Multiple Instruction Single Data, many processors execute instructions over the same data; and (4) <u>MIMD</u> - Multiple Instruction Multiple Data, these computers execute several independent programs at the same time. In the late 1970's Denelcor, Inc. developed a machine architecture known as the heterogeneous element processor. The HEP computer was a multiple instruction multiple data (MIMD) machine. Also in the late 1970's a new style of instruction level parallelism was called Very Long Instruction Word (VLIW). (Lazou 1986, 54)

Supercomputers in 1980 incorporated large scale integration (LSI) for both memory and logic circuits. This later led to VLSI (Very Large Scale Integration). The

Control Data Star-100 was refitted with LSI chips. This led to the production of the Cyber 205. Cray Research at this time put their effort into the production of the Cray-2 in 1985 and the Cray X-MP series. The Cray is an extension of the Cray-1 architecture. "The multiple CPU organization of the Cray X-MP computers significantly increases the opportunity for parallel operations during the solution of a large scale problem" (Lazou 1986, 2). This is accomplished by using the multitasking technique.

Multitasking and multiprocessing are two approaches used to increase system performance and response. Many programs or tasks appear to operate in parallel in a multitasking system. The responsibility for each task can be assigned to a specific part of the system so that the entire system works like a team. The operating system supplies intertask communication and scheduling. A multiprocessing system uses a number of processors, each with its own memory to execute programs. At the same time, each processor has access to a common memory. This allows execution of different parts of the same program simultaneously. The advantage of multiprocessing over multitasking is that separate programs can be assigned to specialized functions. In 1984, published articles stated Japan was going to bypass the U.S. in the manufacture of supercomputers. A couple of supercomputers manufactured at this time in Japan were the Fujitsu FACOM VP-200 and the HITAC S-810. The FACOM VP-200 compared its results to the U.S. Cray X-MP. One of Japan's weaknesses at this time was dependence on IBM software. As of March 1985 there were 150 supercomputers in the western world. Ten were made in Japan, the rest were made in the United States. In 1985 supercomputers were selling for between $10 million and $20 million each. (Davidson 1993, 27)

In October, 1991, Thinking Machines Corporation introduced the Connection Machine model CM-5. It has MIMD capabilities and uses a RISC (Reduced Instruction Set) microprocessor. It also uses a Unix-like operating system. Danny Hillis, a student at Massachusetts Institute of Technology, is one of the co-founders of Thinking Machine. This company was a pioneer in massively parallel computing. Thinking Machines Corporation declared bankruptcy in August, 1994. They netted $83 million in revenue in 1993, but lost $40 million the last two years. As a contributor in the evolution of supercomputers, Thinking Machines Corporation was eventually "squeezed out" by bigger companies in the computer market. (Schmit 1994, 2B)

NCUBE hardware architecture is a scalable network between 32 and 8192 general purpose processing nodes. The NCUBE system can perform 60 billion instructions per second and 27 GFLOPS (billion floating-point operations per second). Also developed in the 90's was iWarp which was a joint effort between Intel Corporation and Carnegie Mellon University. Also developed by Intel was iPSC and iPSC/2. (Lafferty 1993, 60)

Computing in parallel has been done on shared memory systems, distributed memory systems, and even a network of workstations. These different architectures are programmed using a number of different paradigms. "There are three basic paradigms for parallelism. They are the program's *result* (divides the total number of work into portions to match the number of processors), a program's *agenda of activities* (master processor sends work to slaves who keep coming back for more when the first job is done), and an *ensemble of specialists* (specific processors perform specific types of work)" (Carriero 1992, 14). A large number of parallel programming languages and models have been introduced during the last decade. Established languages include C and Fortran. Other languages are Linda, Occam, Ada, and Parallaxis. Object-oriented languages are Smalltalk, Simula, and C++. Programmers can't really agree which parallel programming language is best. There are at least ten different languages. Running applications today in parallel is easier thanks to advanced multithreaded Unix kernels and other tools according to Chuck Narad, engineer at Sun Microsystems Computer Corporation. (Narad 1993, 52)

Supercomputers are best used for scientific problems and calculating numbers. Current supercomputers have had an impact in the area of science and engineering including nuclear engineering, fusion devices, aerodynamic designs for weapon and satellite delivery systems, research in astrophysics, elementary particle physics, quantum chemistry, crystallography, oceanography, and meteorology. Commercially supercomputers are used in aircraft design, motor car design, film production, seismic oil, and mineral exploration. Future uses of supercomputers remain to be seen, but based on the changes and advances made thus far, the sky's the limit. (Lazou 1986, 2)

Mr. Grady Booch reported in an article in IEEE Software that he remembered seeing the science fiction movie, *2001: A Space Odyssey* in the late 1960's while growing up. He went on to report that Arthur C. Clarke and Stanley Kubrick foresaw a number of future technologies in their movie, such as massive parallel machines, videoconferencing, networks that seem to be everywhere, and voice recognition. Upon leaving the theater he thought to himself, "Who is going to write all that cool software?" (Booch 1994, 33).

At the secondary level instructors need to continue to motivate their students to pursue careers in computer science. Parallel processing seems to be a tool that can be utilized to accomplish this task.

Materials and Methods

Design of the Investigation

The cognitive ability of high school students to understand and implement parallel processing was studied. A group of eight high school students consisting of three boys and five girls participated in the study. Four of the students were juniors and four were seniors. The participants in the study have average and above average GPA's.

The project was conducted over a three week period with the class meeting for 42 minutes each day. If a teacher wanted to spend more time they could include more instructional modules or increase the learning activities in each module. The project was conducted during the month of May. The students completed two semesters of QBasic prior to this project.

Software and Hardware Specifications

The two pieces of software needed for this project are the P4 parallel language and the Linux operating system. Argonne National Laboratory developed P4 for programming a variety of parallel machines. P4 is a library of macros and subroutines. Before P4, programmers used the m4-based Argonne macros system "...described in the Holt, Rinehart, and Winston book Portable Programs for Parallel Processors, by Lusk, Overbeek, et al" (Butler 1992, 1). This is how P4 got its name. The present P4 system uses the computational models: monitors for the shared-memory, message passing for distributed memory, and support for combining the two. P4 is supposed to be portable, simple to install and use, and efficient. P4 can be used to program workstations on networks, advanced parallel supercomputers, and single shared memory multiprocessors. Some machines it is currently installed on are Convex, Cray X/MP, Sun, DEC, Silicon Graphics, HP, IBM RS6000, nCube, Intel IPSC/860, and Thinking Machines CM-5. The current release is version 1.4. Questions about P4 can be e-mailed to p4@mcs.anl.gov. One can obtain the complete distribution of P4 by anonymous ftp from info.mcs.anl.gov. The manual and installation files can be found in the 'p4/doc' directory. It comes with example programs on shared memory, message passing in C, message passing in Fortran and others. (Butler 1994, 2)

Linux is a free UNIX clone for personal computer systems. It supports full multitasking, the X Window System, TCP/IP networking, Emacs, UUCP, mail, and news software. Linux is the operating system kernel that does process scheduling, virtual memory, file management, and device I/O. It will run on any 386, 486, or pentium computer.

UNIX is one of the most popular operating systems because of its large support base and distribution. It was developed in the mid 1970's as a multitasking system for minicomputers and mainframes. Versions of UNIX exist for everything from personal computers to supercomputers. The current U.S. cost for UNIX is about $1500.

Linux is a freely distributable version of UNIX developed by Linus Torvalds at the University of Helsinki in Finland. Many programmers across the Internet also helped with its development. It first started as a hobby for Linus who was motivated by a small UNIX system called Minix. Early development of it dealt with task switching features of the 80386 interface all written in assembler. The first official version of Linux was introduced on Oct. 5, 1991 (version .02). The official release of version 1 that was supposedly complete and bug-free was March 1992. As of December 1993 the kernel was still at version 0.99. Berkeley UNIX played an important role in helping to develop Linux. (Welsh 1995, 6)

Some of the people who use Linux are application developers, network providers, kernel hackers, and multimedia authors. Other uses Linux has are "managing telecommunications and data analysis for an oceanographic research vessel, research stations in Antarctica, and at hospitals to maintain patient records" (Welsh 1995, 6).

Over networks Linux is capable of sharing files, supporting remote logins, and running windows on other systems. It is a complete multitasking, multi-user operating system. The source code for Linux includes the kernel, device drivers, libraries, user programs, and development tools.

The hardware components needed needed for this project are two computers, two hard drives, two ethernet cards, and a connecting cable to connect the two processors. A CD-Rom drive is convenient for installing the software. The hardware needs to be fine tuned to support the software. This may include setting IRQ's for the various boards on the system.

Procedures

To conduct this study, a series of six instructional modules were used over a three week period. In the first insructional module the learning activities were followed as they are written [See Appendix A]. For the human simulation activity, a random number generator program was written by the instructor to generate 80 random numbers. These were put in an array and printed out [See Appendix B]. A log sheet to record single and multiple processor times was prepared by the instructor [See Appendix B]. The students were told they were going to each simulate a single processor and calculate a total for the list of eighty random numbers. They were first asked to enter on their log sheet an individual hypothesis for the amount of time it would take to accomplish this task. The students were timed using calculators to total the random numbers. They entered their individual times on their log sheets. For the multiple processor simulation, similar steps were followed, except now they were to work as a team. During the actual event each student had to add ten numbers. The first person's summation became the master process, while the others were slave processes. The slaves had to run their totals to the master process which, in turn, calculated a grand total. The timing of the simulation stopped at this point. Average times were calculated for the group for each set of hypotheses and each set of actual times.

The second instructional module began with a lecture discussion on how parallel processing has evolved over the decades [See Appendix A]. It was a general overview in order to convey to the students just how much parallel processing has developed from the late 1940's to the 1990's. The students were given a parallel processing program written in P4 [See Appendix B]. They were asked to look for code that was similar to QBasic code they already knew. The code they found was discussed and analyzed. This led to the programming language C, since P4 is written in C.

An in-depth discussion on the C programming language was the main focus in instructional module three [See Appendix A]. A handout was give to the students called "A Guide To C For Pascal Programmers" [See Appendix C]. These students are not Pascal programmers, but the handout covers the main topics needed for this instructional module. Specific sections of the handout were discussed in detail. They were the basic program structure; statements and compound statements; data types a, b, c, and e; the first four operators; how to increment and decrement an integer; control structures; input and output; preprocessor; compiling a program; and executing a program. The class applied their handout notes on C to a program written in C [See Appendix C]. With the guide as a reference it was much easier to go through the code and analyze it.

Instructional module four followed the learning activities listed [See Appendix A]. After a brief review on compiling and executing C programs, the students used the computers with the Linux operating system. They took turns compiling and executing the random number C program handed out in instructional module three [See Appendix C].

The learning activities in instructional module five clearly explain what was done over a three day period [See Appendix A]. The handout of UNIX commands can be found in Appendix D. The instructor logged into the University of Illinois at Springfield through the modem to allow the students to experience a larger UNIX system.

The procedure for the sixth instructional module includes giving the students a handout on P4 notes [See Appendices A and E]. The code the students typed into P4, as indicated in the learning activities, is in Appendix E. The other two parallel programs they compiled from the hard drive. They can be seen in Appendices B and E.

Methods for Observation and Interpretation

Students were visually observed on a daily basis. Oral questioning and discussion occurred between the instructor and students to aid in the evaluation process. This helped to indicate when further instruction on a concept was needed. Different instructional modules were videotaped for documentation purposes as well as for review of the instructional lesson. Students commented on the tape about C code that looked similar to QBasic code they had used in class.

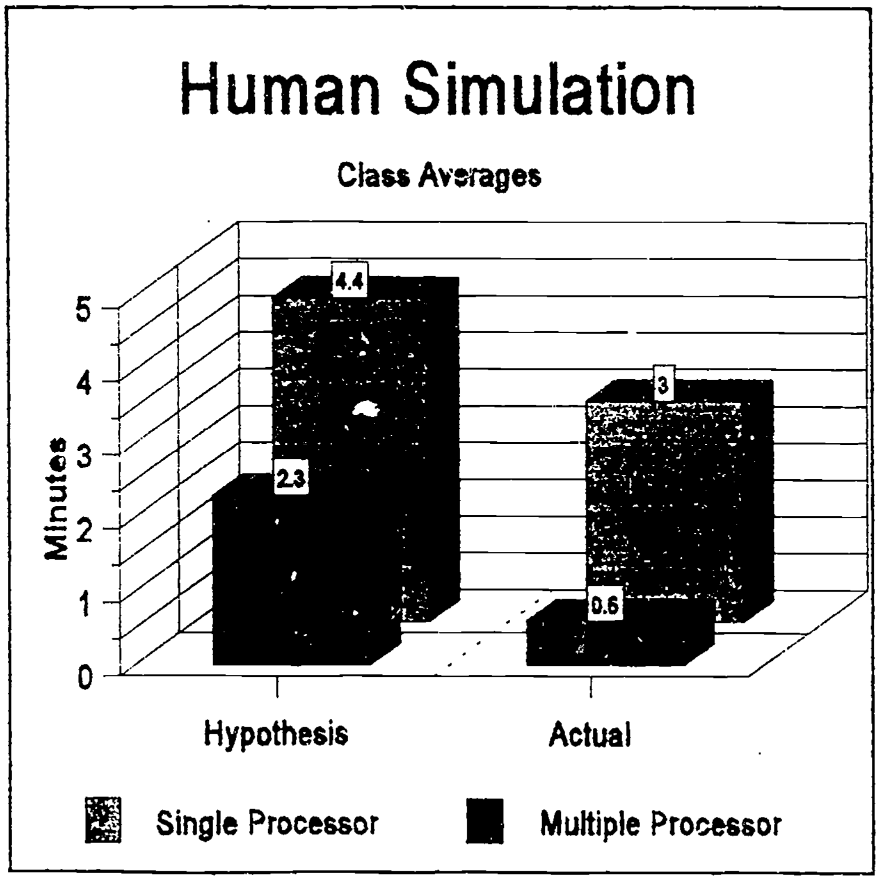
In order to evaluate if the students were comprehending, a review day was included followed by a quiz day over C at the end of its instructional modules [See Appendix F]. This was also done for UNIX/Linux [See Appendix F]. Due to a lack of time the quiz over P4 was omitted. This instructional material was included on the final evaluation test. We concluded the evaluation process with a final test over all three concepts [See Appendix F]. Statistics were correlated for each of these post-evaluation measures. Students filled out an unsigned opinion survey of the whole project after the final test [See Appendix G]. Statistics were recorded for this as well. The statistics are displayed in the Result Section.

Results

Overall the results of the instructional modules clearly indicate that high school students can understand and implement parallel processing. Some of the results surprised not only the students, but the instructor as well. The graphs in this section show the results of the observations and experimental design.

The results of the human simulation are shown in Figure1. There is a significant change between the single processor actual time class average and the multiple processor actual time class average. The single processor actual time average is rather high due to one student losing her place in the list of random numbers. This skewed the average.

Figure 1



The results of students searching P4 code for code that was familiar to them, given they had studied QBasic, led the class into the next instructional module on the language of C. The C handout guide gave the students concrete examples about the various parts of the C language. The guide was helpful in analyzing the random number C code. Hands on experience is the best teacher as was proven when the students compiled and executed C programs. Students concluded their instruction on C with a quiz (Appendix F, p70 & 71). Large amounts of information was given on C, which overwhelmed the students taking the quiz. The instructional module was re-evaluated and an extra day was spent on C. The students were better prepared on the second quiz. The scores were: 100, 93, 93, 93, 86, 86, 79, and 64. The statistical results can be seen in Figure 2.

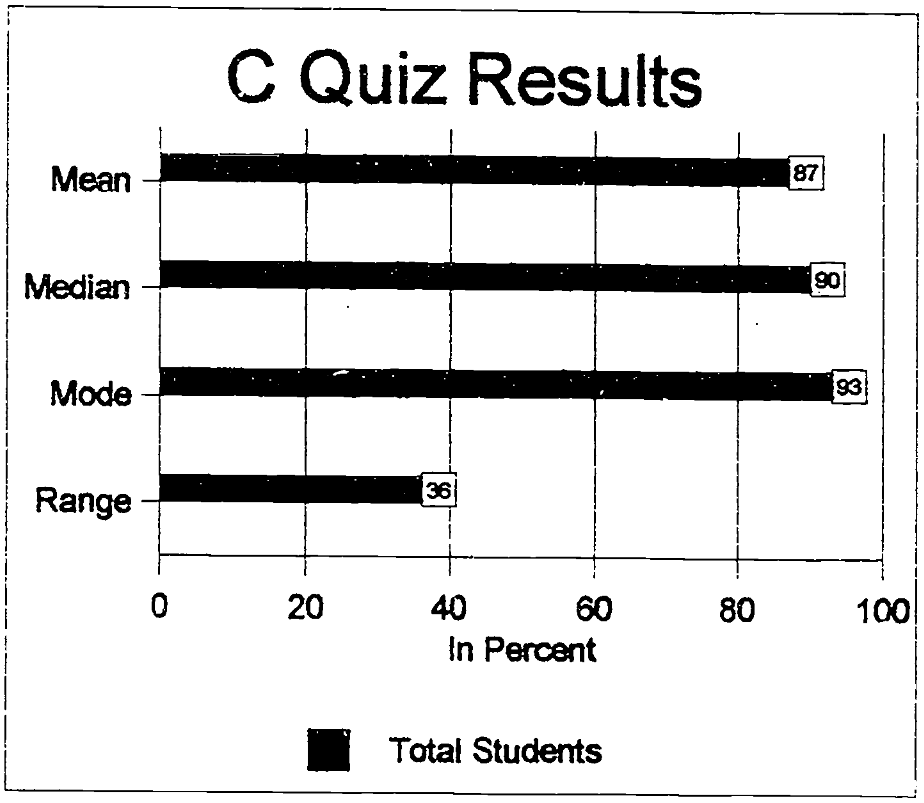


Figure 2

The Linux handout was very useful when applying the commands on the computers in the lab and when the students were logged in to the University of Illinois at Springfield's Computer Lab through the modem. The students really enjoyed Linux and the e-mail available with it. The results of the Linux/UNIX quiz (Appendix F, p72) can be found in Figure 3. The scores were: 100, 100, 90, 90, 90, 90, 90, and 90.

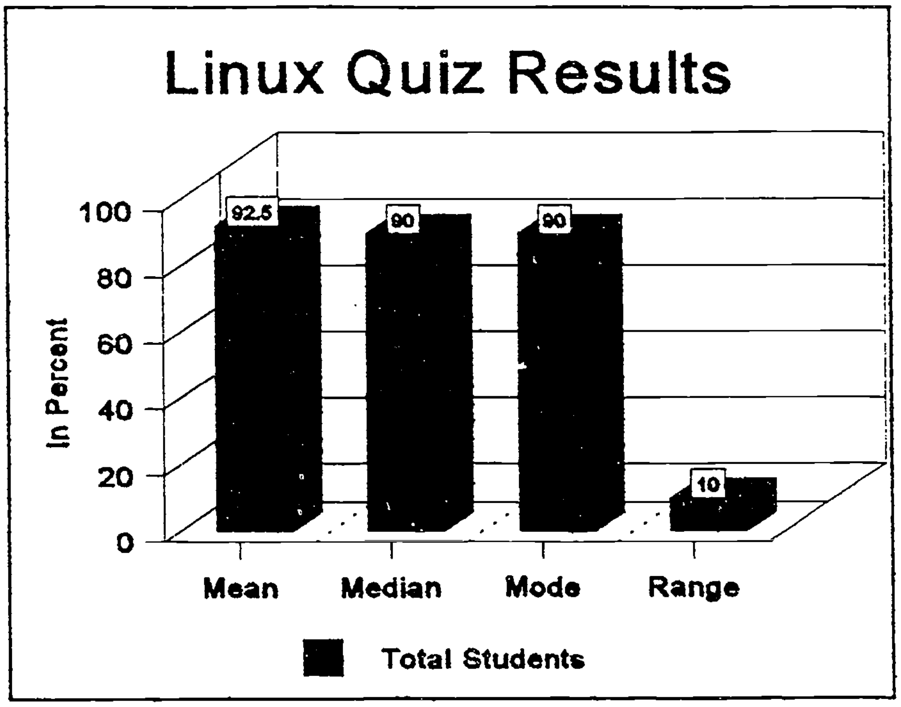


Figure 3

As with C and Linux, hands on activities enhanced the learning of P4. The time spent on the computers for all of the learning activities solidified the concepts that were in the students minds. Three P4 modules were run on the parallel processors. The cognitive understanding of P4 was evaluated on the final test given over all three concepts.

The results of that test (Appendix F, p73-76) can be found in Figure 4. The scores were: 95, 94, 93, 92, 89, 87, 80, and 77.

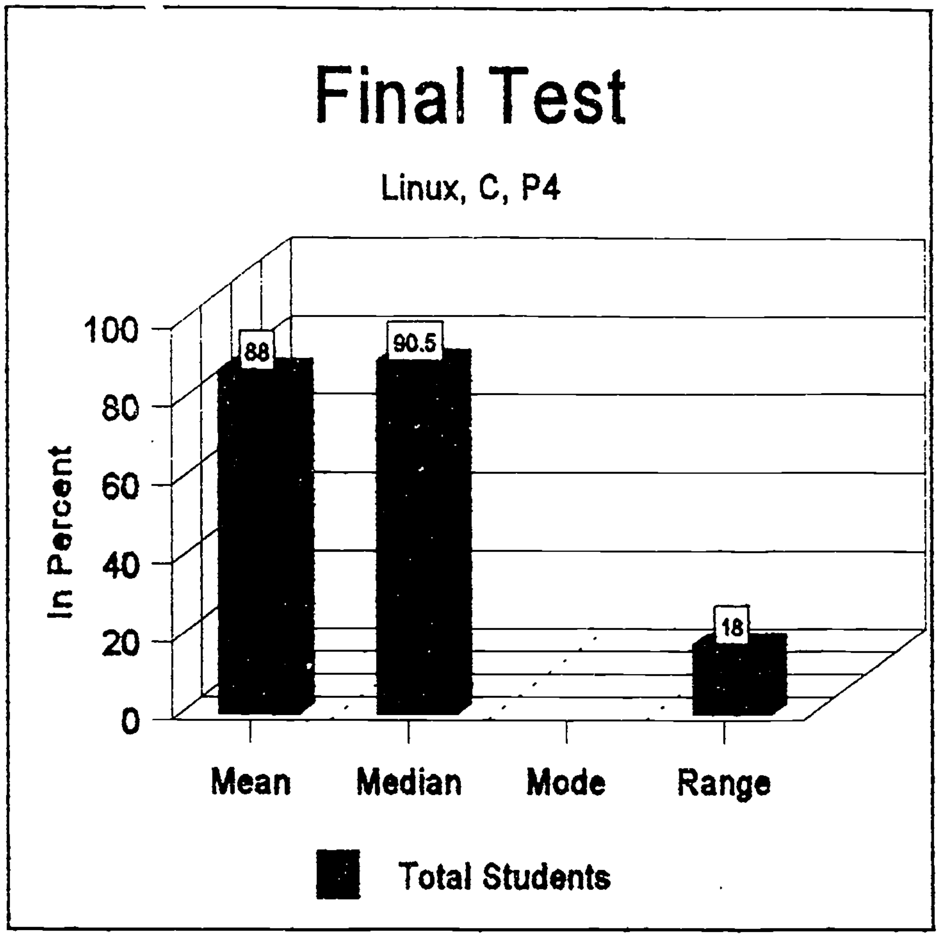


Figure 4

After completing the final test the students filled out a survey (Appendix G, p77) giving their opinions on all of the content areas that were taught. The results of the survey can be found in Table 2. The last column indicates the average for each question. It is based on adding the totals for each question and dividing by eight, since eight students took the survey.

Student Survey Results					
	Excellent(4)	Good(3)	Fair(2)	Poor(1)	Avg/Quest
Question 1	8	18	0	0	3.25
Question 2	16	6	4	0	3.25
Question 3	16	6	4	0	3.25
Question 4	16	6	2	1	3.125

Table 2

Thirty high schools in central Illinois were surveyed about programming and parallel processing. Twenty-one of the surveys were returned (a 70% response rate). Both large and small schools were surveyed. The results . f the survey can be found in Table 3. The last column in the table indicates the most frequent answer given. Some of the questions totals do not add up to twenty-one due to the fact that some respondants did not answer all of the questions. The majority of the large schools responding did indicate teaching programming in at least one language. The majority of the small rural schools either did not offer programming in any language or they did not have it presently due to a insufficient enrollment for those classes. Only a small number of rural schools currently teach programming. None of the schools responding taught P4 or parallel processing in another language. The survey can be found in Appendix G, p78.

The following questions were asked:

1. Do you teach programming in your computer science curriculum?

   A. Yes B. No

2. How many programming languages do you teach at your school?

   A. 1 B. 2 C. 3

3. What programming languages do you teach in your computer science curriculum? Circle all that apply.

A. Basic/QBasic B. Pascal C. Cobol D. Fortran

E. C/C++ F. P4

4. Have you ever heard of parallel processing?

A. Yes B. No

5. Do you teach P4 as a parallel processing language?

A. Yes B. No

6. Would you be interested in learning more about parallel processing in the future?

A. Yes B. Maybe C. No

7. Would you consider at some point in the future teaching parallel processing?

A. Yes B. Maybe C. No

H.S. Teacher Survey Results							
	A.	B.	C.	D.	E.	F.	Maj
Question1	15	6					yes
Question 2	9	7	1				1
Question 3	12	10	1	1			Basic
Question 4	8	12					No
Question 5	0	19					No
Question 6	4	10	6				May
Question 7	2	13	5				May

Table 3

Discussion

The results definitely indicate that high school students are capable of understanding and implementing parallel processing. These results also indicate a high level of understanding in the areas of the Linux operating system and the programming language C.

The instructional module on C needed to be lengthened by one day in order to review more thoroughly for the first quiz. This also involved repeating the quiz. The students were insecure as to what kind of questions might be asked, even though examples of questions were verbalized in class. The first C quiz ended up being a pre-quiz, which is OK. This was their first exposure to these more advanced programming concepts.

The hands on experience with the modem and Linux/UNIX was the solid experience necessary to emphasize the basics of UNIX to the students. The students had no problems with e-mail once they started. The joe editor in Linux was easliy adapted to.

If the study were repeated it would be more beneficial to lengthen the P4 instructional module by another week. This would provide more hands on experience with additional programs and would build upon the knowledge base already established.

From the teachers surveyed it was found that additional categories needed to be added in order for each question's results to total the number of people participating in the survey. For example question number two needed response D for zero languages taught.

Another example is in question numbeı three where response G should indicate other languages taught. It was assumed that the programming languages taught at high schools were done on IBM or IBM compatible personal computers. One respondent indicated teaching hypertalk.

This topic definitely warrants further study. High school students are ready for more challenges in the area of computer programming. It is up to educators to provide those learning opportunities.

Appendix A

INSTRUCTIONAL MODULE

LESSON ONE

Learning Objective:

Define parallel processing, give analogy examples of parallel processing, and do a hands on simulation.

Time: 1 - 1 ½ class periods

Equipment Needed:

Word processor software on computer to take notes, calculator, and pen or pencil.

Learning Activities:

1.) Through lecture define parallel processing

2.) Give analogy example 1 of one person putting books on library shelves vs. many people working as a team passing books along from one section to another until all books are shelved.

Give analogy example 2 of passing numbers down a pipeline until they are sorted in order.

3.) Do an introductory activity that is a human simulation of single processors adding a group of random numbers and a second one simulating parallel processors in which the students work together as a team.

Appendix A

INSTRUCTIONAL MODULE

LESSON TWO

Learning Objective:

Give a history overview of parallel processing and compare P4 program code with QBasic code that is already known.

Time: 1 class period.

Equipment Needed:

Word processor software on computer to take notes, P4 program code handouts.

Learning Activities:

1.) Lecture on general history of parallel processing.

2.) Go over P4 code looking for familiar code. Explain the use of C programming language with P4.

Appendix A

INSTRUCTIONAL MODULE

LESSON THREE

<u>Learning Objective:</u>

Introduce C programming language and evaluate random number C code for a single processor.

<u>Time:</u> 1 class period

<u>Equipment Needed:</u>

C program handout

Random number on single processor handout.

<u>Learning Activities:</u>

1.) Discuss C program handout

2.) Go through random number code line by line.

Appendix A

INSTRUCTIONAL MODULE

LESSON FOUR

Learning Objective:

Compile random number C code on Linux system.

Time: 1 class period

Equipment Needed:

Computers with Linux operating system installed on hard drives

Learning Activities:

1.) Go over from C notes how to compile and execute C program code.

2.) Instruct students how to log on to Linux system.

3.) Compile and execute random number code on single processor. Make sure each student gets a chance to compile and execute the code.

Appendix A

INSTRUCTIONAL MODULE

LESSON FIVE

Learning Objective:

Introduce Unix/Linux in more detail.

Students experience hands on activities.

Time: 3 class periods.

Equipment Needed:

Unix/Linux commands handout.

Computers with Linux operating system installed on hard drives.

Access to a modem.

Learning Activities:

1.) Handout Unix/Linux commands most commonly used to students. Go over and discuss them.

2.) While students are at the computers, go over the Joe editor using our random.c code to edit. Illustrate how to change code, re-compile it, and go to specific line numbers in Joe editor to correct mistakes.

3.) Explain how to use the e-mail (electronic mail) with the Linux system. Allow students to send each other e-mail.

4.) Login to the University of Illinois at Springfield computer lab through the modem and allow students to practice commands from handout to get a better idea how they work on a larger Unix system.

Appendix A

INSTRUCTIONAL MODULE

LESSON SIX

Learning Objective:

Introduce P4 commands and run P4 code in parallel.

Time: 3 class periods

Equipment Needed:

Parallel Processors side by side with Linux and P4 installed on the hard drives.

Also need P4 handout for students and P4 code to enter or already loaded on the hard drives.

Learning Activities:

1.) Handout P4 notes and go over special P4 commands that will be used.

2.) Have students enter code for "tiny" P4 program that passes the id number of the master process to the slave, who in turn prints a message with the id number in it. The slave processes do not do any work.

3.) Have students "make" and execute the P4 program that uses message passing to send a character to the slave process and increment it.

4.) Have the students run the P4 program that creates random numbers stored in an array and sends them to be added by the slave processes while the master waits for the return total sum.

LOG SHEET FOR PARALLEL PROCESSING

Single Processor Human Simulation

Enter your hypothesis for the time it will take to total the attached list of random numbers.

Hypothesis:

Enter the actual time it took to total the list of random numbers.

Actual Time:

Multiple Processor Human Simulation

Enter your hypothesis for the time it will take to total part of the attached list of random numbers and then give that total to the first person done to finish the final total.

Hypothesis:

Enter the actual time it took to total part of the list of random numbers and the final total done by the first person finished.

Actual Time:

Appendix B

```
'Program to generate random numbers
'Spring 1995
'
DIM Numarray(1 TO 80)
RANDOMIZE (100)
Low = 0
High = 100
FOR I = 1 TO 80
r% = INT(RND * (High - Low + 1) + Low)
Numarray(I) = r%
PRINT Numarray(I),
NEXT I
END
```

```
65             78             62             88             42
75             11             40             3              63
55             10             17             70             17
2              32             80             50             43
22             0              37             60             60
50             29             85             5              73
100            79             35             79             28
83             79             3              8              97
96             3              82             79             67
66             95             72             17             47
80             58             60             39             66
32             66             60             9              4
61             64             55             26             46
82             13             44             39             86
73             74             51             79             33
0              76             8              57             12
```

Kathy Sheary
P4 System Message Passing
Message Type Character

```
/* ---------------------- msgchar.c -------------------- */

/* This program sends a value of A to the first slave each
  slave increments the value and the final slave returns it
  to the master. The master receives the final value and
  stores it in letter and prints it.
  The purpose of the program is to demonstrate how to send
  and receive character values using the p4 system
  Ted Mims

  Springfield, IL 62794

*/
#include "p4.h"
int main(argc,argv)
  int argc;
  char **argv;
{  void master(), slave();
  p4_initenv(&argc,argv);
/* Of all of the processes which execute this code, only one has
  p4_get_my_id = 0.  This will be the commanding process.  All
  others will be independent, subserviant processes.
*/

  if (p4_get_my_id() == 0)
    { p4_create_procgroup();
     master();
    }
  else
    slave();
  p4_wait_for_end();
  printf(" Main driver exiting normally for process %d.\n",p4_get_my_id());
}
/* constants used to define data types of messages */
#define TAGINIT 10
#define TAGDATA 20
#define TAGEND  30
```

Appendix B

```
/* The master will initialize all of the slaves and send the value
  of one to the first slave. The master will wait to receive a
  final value that has incremented by each slave
*/
void master()
{  int from, i, to, type, size, num_total_slaves;
/* type-- used to indicate the type of message
  to -- used to indicate which slave to send to
  size -- used to indicate the size of the message
  num_total_slaves -- used to hold the number of salve processes
  form -- used to hold id of slave to receive from
  i - used for loop control
*/
  char *msg, letter;
/* msg -- used to hold pointer to the character message
  letter -- used to store the char value received from slaves
 */

  size = sizeof(int);
  num_total_slaves = p4_num_total_slaves();
/* initialize all slaves */
        for (i = 1; i <=  num_total_slaves; i++)
                p4_sendr(TAGINIT,i,(char *)&i,size);
        p4_dprintf ("\n master initialized all slaves");
/* send 1 to process 1 */
        to = 1;
      letter = 'A',
      size = sizeof(char);
        p4_sendr(TAGDATA,to,(char *)&letter,size);
        p4_dprintf("\n master sent A to first slave");

/* wait for reply to come from last processor */
        type = TAGDATA;
        from = num_total_slaves;
      msg = NULL;
/* -1 to receive from any slave */
        letter = ' ';
        p4_recv(&type, &from, &msg, &size);
        letter = *(char *)msg;

        p4_dprintf("\n received letter %c from last slave ", letter);
        p4_msg_free(msg);
    p4_dprintf("\n master exiting normally\n");
 }
```

Appendix B

```
/*
Each slave will receive an integer from its left neighbor, add one
to it and pass it on to the next slave.
*/
void slave()
{  int next, type, sender, size, id, num_total_slaves, i;
/* int_value -- used to store received msg as an integer
  next -- used to indicate next slave or host to send to
  type-- used to indicate the type of message
  sender -- used to indicate the id of the sending process
  size -- used to indicate the size of the message
  id -- used to hold the id of this process
  num_total_slaves -- used to hold the number of salve processes
  i - used for loop control
*/
  char *msg, letter_value;
/* msg -- used to hold pointer to the character message
  letter_value -- used to store received msg as an integer
 */
  size = sizeof(int);
  type = TAGINIT;
  sender = 0;
/* clear out value stored in the msg */
  msg = NULL;
  id = p4_get_my_id();
  num_total_slaves = p4_num_total_slaves();
  if (id == num_total_slaves)
    next = 0;
  else
    next = id + 1;
/* Each process accepts initialization message */
  p4_recv(&type,&sender,&msg,&size);
  p4_msg_free(msg);
  p4_dprintf("\n process %d initialized\n", id);
  msg = NULL;

/* Receive an message from the left neighbor */
  sender = id - 1;
  type = TAGDATA;
  p4_recv(&type,&sender,&msg,&size);
/* convert the received message of type char to type integer */
  letter_value = *(char*)msg;
  p4_msg_free(msg);
  p4_dprintf("\n process %d received letter %c from left neighbor",
   id, letter_value);
/* increment letter_value by 1 */
  letter_value++;

/* send to right neighbor */
  p4_sendr(TAGDATA,next,(char*)&letter_value, size);
  p4_dprintf("\n process %d terminating",id);

}
```

Appendix C

A Guide to C for Pascal Programmers

This is written to help Pascal programmers translate their programming style into a C programming style. This is not necessarily complete, but all effort has been made to make this understandable. For further information, the C reference paper is one place to look.

1. Basic Program Structure

The basic structure of the program is this.

```
include declarations
preprocessor definitions
global declarations and definitions
main()
{
  variable declarations
  program statements
}
```

The only lines above which must be written exactly as is are the "main" line and the two lines with braces.

One major difference between Pascal and C is that in C all procedures and functions are really functions, and all exist at the same level of nesting as the main function, so that the variable scope of Pascal is nonexistent. Variables declared in the main program are known only in the main program; variables declared in other functions are known only in those functions in which they are declared (except for global variables, which will be mentioned later).

2. Statements and Compound Statements

The statement-compound statement style is much like Pascal However, the open and close braces ({}) are used instead of "begin" and "end". Comments are begun and ended as in PL/1. That is, /* precedes and */ follows a comment. Statements and comments can extend over many program lines, just as in Pascal. Anywhere a statement can appear, a compound statement can also occur. Semicolons must terminate any statement.

3. Data Types

a) "int" is the data type for 32-bit twos-complement integers

b) "char" is the data type for characters

c) "float" is the type for 32-bit floating point numbers

d) "double" is for 64-bit (double-precision) floating point

e) "char array[20]" will declare an array of characters of length 20.

A Guide to C for Pascal Programmers

f) Matrix arrays are declared in a slightly odd way:

"int mat[15][35]" will declare a matrix of 15 rows and 35 columns, corresponding to a Pascal declaration "integer mat[0..14,0..34]".

NOTE--all array subscripts start from zero, so this declaration would be for an array of length 20 with subscripts 0 through 19. This can be confusing--I myself almost always declare arrays to be one longer than necessary and never use the zero-th location. Structured data types are declared in a manner similar to Pascal. The definition of a structure data type called "mpnum" (for multiprecise number) could be

```
struct mpnum
{
int len;
int digits[20];
}
```

and then the declaration of a variable "thisnumber" of type "mpnum" would be struct mpnum thisnumber;

It is also possible to define the type and declare a variable in the same statement. I would recommend, however, that you define the type globally, and declare the variables inside each routine.

4. Operators and Expressions

The assignment operator is the equal sign. It is possible to declare and initialize variables in the same statement, as in "int var = 0;".

In testing variables in expressions for use in while loops, for loops, and if statements, the operators are

== for testing equality, as in "if(i==1)"

!= for testing inequality

>= and <= for greater and less than, respectively

> and < for greater and less than, respectively

&& and || for boolean anding and boolean oring, respectively

Now, for two of the strange and beautiful bits of C. You will frequently have need of things like

```
i = i+j;
```

In C, this should be written

```
i+=j;
```

The difference is that for "i=i+j" the variable i will be fetched twice, once on the right hand side and once to know where to store the result. The "+=" notation does only one fetch. Although efficiency is not necessarily a serious consideration, this is considered the "proper" way to write C code. Also, it requires less text, and therefore can be easier to read since there is less to read. If what you have to do is just increment or decrement an integer variable, though, use "i++" or "i–". See the stuff on for loops for an example.

5. Control Structures

The while loop is, for example,

```
While(donewithloop == NO)
{
   body of loop statements fit in here
}
```

Note that, as in Pascal, if the body of the while is only one statement, then the {} can be omitted. This is true for all the control structures. The for loop is different from other languages, for example:

```
for(beginning value; condition; what to do each time)
{
  statements
}
```

Appendix C

A Guide to C for Pascal Programmers

Example (note the i++):

```
for(i=1; (i<=20) && (doneloop == NO);i++)
{
  statements
}
```

This runs a loop from 1 to 20, [illegible]rementing by one each time, with the added condition(this is a nice feature of C) that a flag variable can also be tested to get out of the loop early. The "and" in expressions is done with a "&&", and the "or" is done with a "||". Be very careful with combined expressions, as the order of evaluation in C is different from Pascal. I always parenthesize everything completely to avoid learning the rules.

The if is very simple:

```
if (condition)
{
  statements
}
else
{
        statements
      }
```

The usual rules about leaving off the else if not needed and doing else ifs apply in C.

There is also a "switch" statement in C, which corresponds to the "case" statement.

6. Input and output

The "read statement is called "scanf" in C. To read in an integer num, a floating point x, a character inchar, and a "string"(array of characters)str, one would execute

```
   scanf("%d %f %c %s", &num, &x, &inchar, &str);
```

The %d stuff is the input format. NOTE--the ampersand is very important. This means pass the variable by reference--pass the address of the variable--and is the only way to get something back from a C function, which is what the scanf really is. One of the very common errors is to forget this, and you get zapped very quickly after executing the scan.

The "write statement is called printf", as in

```
   printf("num=%6d,x=%9.3f,inchar=%c\n"2,num,x,inchar);
```

Note that the "newline" must be explicitly specified with the backslash n sequence. The %6d prints integers in a field of width 6, and the %9.3f does a floating point number in a 9-space field with 3 decimals. And the rest of the stuff between quotation marks is printed exactly as is. Note that you do not put the ampersands before the variable names in the printf, which is why you often forget them in the scanf.

7. Data Tying

Pascal is a strongly typed language. When you pass integers to reals in subprograms, the compiler catches you. When you overrun array subscripts, you get flagged, C doesn't care about this sort of stuff at all. It is not strongly typed. Indeed, in C it is possible to do most of the "unsafe" things that Pascal was designed to not allow one to do. Some of the common errors are to pass the wrong kind of things to subprograms, or to pass by value when the subprogram has arguments declared by reference, and so forth. (One such error is trying to print integers with floating point format, or floating point values in integer format. The print works fine, simply converting the bit configuration into whatever it corresponds to in the other data type.)

8. Parameter Passing

Appendix C

A Guide to C for Pascal Programmers

The default passing method for all parameters is by value. To call a function testit with two integer arguments a and b, the calling program has

```
testit(a,&b);
```

and the function testit looks like

```
testit(a,b)
int a,*b;
{
  statements of the function
}
```

The *b in the declaration declares b to be the address of an integer variable b. In the function testit, the value of b would then be referred to as *b, read "that which is pointed to by b". This can be cumbersome at first.

All functions in C return(or can return) a value which is by default an integer. The "return(somenumber)"s statement returns a value of whatever "somenumber" is. If no return statement is executed, then the returned value is "void". Functions which return values other than "int" must be declared as such both in the calling program and in the first line of the function itself (just like FORTRAN). No compile-time errors will occur(no strong data typing, remember) but the bit configurations returned will be interpreted as if they were integers.

Note that the use of a returned value can be very useful. The scanf, for example, will return a value called EOF if end of file is encountered. Thus, a read until endoffile can be done as

```
while(scanf("%d", &a) != EOF)
{
  something
}
```

This allows you do the read and the test for end of file in one very clean while statement.

While I'm on that topic, let me mention that you can do assignments inside the condition part of a while loop or for loop or if statement. A structure that will read and print one character at a time until the newline is found is char c;

```
while((c=getchar()) != '\n')
     printf("%c",c);
```

This ties the function call to getchar, which is built in, assigns the returned character to the variable c, and then tests to see whether that is the builtin EOF marker. This can simplify input of character data. There is no endofline or endoffile marker the characters read are the characters there, including all the newline character.

A common mistake in parameter passing is to pass an address to one function and then pass the address to a second function which is expecting the value.

9. The preprocessor

Rather than have defined constants, as in Pascal, C has a preprocessor step. Go back to the first part of this, where it mentions "defines". Before the "main" line, but afte. [illegible]e "include" lines, you can define symbols and character strings to the preprocessor. Before the compile, all the defined symbols are replaced, I usually open all C programs with

```
#include<stdio.h>
#define TRUE 1
#define FALSE 0
#define YES 1
#define NO 0
main()
```

Appendix C

A Guide to C for Pascal Programmers

This will define TRUE,FALSE, and so forth, internally, and when I do tests or return good or bad flags from functions, I return one of these. This way I never need to remember whether true is internally 0 or 1. There are clever things you can do with C by knowing such things, but I regard that as arcane and undesirable style. It is customary to use capital letters for all preprocessor symbols. You can use any sort of symbol you want. A #define TEST <= would define TEST to be less than or equal, and you could use TEST in a condition, to allow you to change the kind of comparison later. Preprocessor stuff will not, repeat not, work inside quoted strings(this often happens in using printf and scanf).

One useful preprocessor feature is this:

```
   #ifdef DEBUG
debugging print statements
   #endif
```

If, when you compile this, you issue the command

```
   cc-DDEBUG programname.c
```

then the statements between the ifdef and the endif are compiled, If you compile it with

Appendix C

A Guide to C for Pascal Programmers

```
cc programname.c
```

then the statements between the ifdef and endif are commented out. This lets you include extensive printing, and then take it out if you want to get rid of extra printout to look at the thing as it really is supposed to run. You can, of course, use any symbol for "DEBUG"; the "-D" compile option defines the symbol "DEBUG" and you can use more than one such option and symbol in a single compilation.

10. The proper Way to Use "make" and to Organize Your Program Files

There is a Unix programming feature called "make" which allows you to break your program into several files and have only the changed files recompiled. There are various ways to use make, What I do, and what I recommend to you, is this.

I have a file called "adefines.c" which has all my preprocessor symbol definitions and definitions of structure data types.

I have a file called "aextern.c" which has the external declarations of global variables which are declared in the main program's heading.

The main program includes "adefines.c" but not "aextern.c" and also has the declaration of global variables.

Appendix C

A Guide to C for Pascal Programmers

The functions called by the main program will be in files called "bsomething.c" and will have "include"s for both "adefines.c" and "aextern.c".

For example, one could have:

File adefines.c:

```
    #include<stdio.h>
    #define FALSE 0
    #define TRUE 1
    #define NO 0
    #define YES 1
struct mpnum
    {
      int len;
      int digits[25];
    };
```

File aextern.c:

```
    extern struct mpnum mpone;
```

File amain.c:

```
    #include "adefines.c"
    struct mpnum mpone;
```

Appendix C

A Guide to C for Pascal Programmers

```
main()
{
   funct1();
}
```

File bfunct1.c:

```
#include "adefines.c"
#include "aextern.c"
funct1()
{
}
```

File makefile:

```
testprog: amain.o bfunct1.o
     cc amain.o bfunct1.o -o testprog
amain.o: adefines.c
     cc amain.c
bfunct1.o: adefines.c aextern.c
     cc bfunct1.c
```

To explain: To compile everything and turn the executable file into something called "testprog" one simply issues the command "make". This causes the makefile to be read. The "testprog" line says that to create "testprog" we need to have "amain.o" and "bfunct1.o". (The suffix in the names is significant. C programs are called "something.c". The compiled object module is then called "something.o".) The second line says that to create "amain.o" and "bfunct1.o" we need to do a "cc" on those files and name the result "testprog" (this is what the -o option does). Continuing down, the amain.o" line says that compiling amain.c requires the "adefines.c" file. Similarly for the "bfunct1.o" lines. Thus, if anything in the dependency tree has been changed, things get recompiled. If not, then only the pieces needed are recompiled. This saves LOTS of time. The prefixing of file names with "a", "b", etc., makes them sort in a nice order when you list file names.

Appendix C

```
/* This is a random number generator in C. It will print each number. Additional code
   may be  added to accumuiate the array of random numbers by the students.
   Kathy Sheary
   University of Illinois at Springfield
    Spring 1995
*/
#include <stdlib.h>
#include <stdio.h>
int rand(void);           /* random built in function   */
void srand(unsigned int); /* srand is used to seed rand */
#define HOW_MANY 15       /* how many random numbers to generate */
#define MY_RANGE 100      /* 0 to MY_RANGE range of numbers     */
#define SEED 55           /* some seed value for using in srand  */
main()
{
 int x, ran;
 srand(SEED); /* seed rand with SEED */
 for (x = 0; x <= HOW_MANY; x++)
  { /* get the remainder of rand() divided by MY_RANGE to store */
   /* a random number between 0 and MY_RANGE in ran          */
   ran = rand() % MY_RANGE;
   printf(" ran is = %d\n",ran);
  }
 }
```

Appendix D

Useful Unix Commands

cat Display a file
cat filename

cc Compile a c program
cc compute.c -- Produces executable file a.out
cc compute.c -o compute.out -- Produces executable file compute.out

cd Change working directory
cd / -- change to systems root directory
cd .. -- change to the parent of this directory
cd -- change to users home directory
cd mydir -- change to the directory mydir if it is a subdirectory of the current working directory
cd mydir/two -- change to the directory two which is a subdirectory of directory mydir

chmod Change access mode of a file or directory
chmod who[operation][permission] file
who u -- user, g -- group, o -- other users, a -- all
operation + add permission, - remove permission, = set permission for specified user
permission r -- read, w -- write, x -- execute,
chmod u+rwx file name
chmod g+rx-w file name
chmod o+r-wx file name

Absolute form of chmod
chmod mode file
mode -- is a bit pattern for the mode of user, group, and others
chmod 754 filename -- 754 represents 111101100 in binary. This is would give rwx permission to the user, rx permission to the group, and r permission to the others.

cp Copy source file to destination file
cp oldfile newfile

Appendix D

Useful Unix Commands

env Status of environment

finger Status of a user
finger userid
finger mims@eagle.sangamon.edu

find find a file by name and print its path
find begin path -name name of file to find -print
find / -name copydots -print
-- This will search for copydots beginning at the root directory and print the path if found.

grep Search for a pattern in a file or inpu to the grep command
ls -l | grep mims --This series of commands will give a long listing of the current directory as input to the grep command. The grep command will find any input lines that contain the string mims and print those lines.

ipcs Display information about interprocess communication message queues, shared memory, and semaphores
ipcs -- Displays information in short form
ipcs -a -- Displays all information

ipcrm Used to remove message, shared memory, and semaphores
ipcrm -m shared memory id
ipcrm -s semaphore id

kill Terminate a process
kill [options] pid-list
options the default option is software termination option 15. To insure the process will be killed use the -9 option.
kill -9 123456 Where 123456 is the process id of a living process.

ls Display information about directories and files
ls [options] filelist
Options

Appendix D

Useful Unix Commands

-a list all files in current directory including
hidden files
-d displays directories without displaying their
contents
-l long form of display seven columns
-x list files horizontally

man Online manual for system commands
man command
man who

mesg Turn message receiving on or off
mesg n -- Turn messages off
mesg y -- Turn messages on

mkdir Make a directory
mkdir mydir -- This command will make a new
subdirectory mydir in the current directory.

more Display a file or input one screen at a time
more filename
ls -a | more -- This group of commands will cause
the output of the ls -a to be piped through the
more command and the list will appear one screen
at a time.

mv Move a file to a new name. Rename a file
mv oldname newname

ps Display process status
ps [options]
Options
-a Display information for all processes
-e Display everything for all processes
-f Display a subset of the -l option
-l Display a long status report 14 columns
-u user Display status for processes owned by the user
ps -u mims -- Displays information about processes
by user mims

Appendix D

Useful Unix Commands

pwd Print name of current working directory

rm Remove a file. Delete a file
rm [options] filename
options
-f -- remove file for which you do not have write permission without asking for your consent.
-i -- interactive You will be prompted for confirmation before the file is actually removed.
-r -- recursive Will remove the contents of the directory and the directory itself.

rmdir Remove a directory
rmdir directory name -- The directory must not contain any files

stty Display or establish terminal parameters
stty [arguments]
arguments too many to list see manual

umask Establish file creation permissions
umask [mask]
mask is an octal number. This number is subtracted from the systems default mask of 777.
umask 066 -- Will leave a permission of 711 for files that are created.
umask can be placed in the .login or .profile files

who Display names of users that are logged on
who

Appendix E

Parallel Processing Notes For P4

In P4, the program runs a master process and slave processes. The master and set of slaves form a ring of processes in which the master reads a message and sends a copy of the message to the first slave, the last slave passes the message back to the master. If the master receives an undamaged copy of the message, it assumes that all went well, and reads another message. The ring of processes is a logical structure in which each process assumes that its predecessor in the ring is the process with the next lower id, and its successor is the process with the next higher id. The master has id 0 (zero) and has the process with the largest id as its predecessor.

The first executable p4 statement in a program should be:

p4_initenv(&argc,argv);

This initializes the p4 system and allows p4 to extract any command line arguments passed to it.

The last executable p4 statement in a program should be:

p4_wait_for_end();

This waits for termination of p4 processes and performs some cleanup operations.

The procedure **p4_get_my_id** returns the unique integer id assigned to the calling process by p4.

The procedures **p4_send** and **p4_sendr** are two of several p4 procedures that are available for sending messages to other processes. They take as arguments the **message type, the id of the process to receive the message, the address of the message(the message itself), and the message length(size).**

The procedure **p4_recv** receives a message from another process and sets the values of all four parameters (same as listed under p4_send). P4_recv will automatically retrieve a buffer in which to place a received message, thus p4_msg_free may be called to free that buffer when it is no longer needed. When -1 is assigned to a parameter like from, then this is a wildcard indicating the slave is willing t receive a message of any type from any process.

The procedure **p4_num_total_slaves** is one of several procedures that the user can invoke to determine information about the current execution.

The statement:

p4_create_procgroup();

reads a procgroup file that the user builds and creates the set of slaves described in the file. This statement must be executed before any slaves can be assumed to exist. This procedure is the method you must use to create processes that do message-passing. The procgroup file describes where all slave processes are to be executed.

Appendix E

The procgroup file we will use on Linux is:

```
local 0
first 1 messages/msg1
#second 1 ~mims/messages/msg1
```

Lines beginning with # are comments. "Local n" where n is the number of slave processes that share memory with the master. The master has no local slaves. The next line indicates the name of the remote machine on which slave processes are to be created. The last line is commented out but set up in the event you want to make it a slave in the future. The name of the file we will be compiling and executing is msg1.

In order to **compile** and link a **p4 program for execution**, you need to make the file. We will need to copy our file, such as lab.c into msg1.c. The next step is to type: make msg1. To run or execute the program type: msg1.

3 Methods for Parallel Programming
1. **Message Passing** - processes running at the same time send & receive messages
2. **Distributed Data Structures** - Processes communicate & coordinate by leaving data in shared objects.
3. **Live Data Structures** - Process is part of data structure.

3 Approaches to Parallel Programs (Relate to Building a House)
1. **Result** - Hand out total project at the beginning to each processor (A house).
2. **Agenda** - Hand out a list of tasks to each processor. (All workers with separate skills work on first thing on agenda, then do 2nd thing, etc.)
3. **Specialist** - Special tasks done by each processor. (Special tasks done by special people: plumber, electrician, etc.)

Appendix E

```
/*
    simplep4.c
    This program is a simple beginning p4 program to illustrate process 0 creating the
    environment.  The procgroup starts the other slave processes. The slave process does
    not do any work.  The slave process is invoked and it prints hello from process 0.
    Kathy Sheary
    University of Illinois at Springfield
*/
#include "p4.h"
main(argc,argv)
int argc;
char **argv;
{
        p4_initenv(&argc,argv);
        if (p4_get_my_id() == 0)
            p4_create_procgroup();
        slave();
        p4_wait_for_end();
}

slave()
{
        printf("Hello form %d \n", p4_get_my_id());
}
```

Appendix E

P4 Code with Array

```
/*
   This program allows the user to pass an array from the main
   processor to a second processor in blocks with a maximum of
   19 integers be sent in each message
   Ted Mims

   Springfield, IL 62794

*/
#define NUMBERS 504     /* size of array to hold numbers for testing */
#define MAX 18  /* MAX is the maximum size for an array of integers */
                /* that can be sent in a message                    */
#include <stdio.h>
#include <stdlib.h>
#include "p4.h"
int main(argc,argv)
   int argc;
   char **argv;
{  void master(), slave();
   p4_initenv(&argc,argv);
/* Of all of the processes which execute this code, only one has
   p4_get_my_id = 0.  This will be the commanding process.  All
   others will be independent, subserviant processes.
*/

   if (p4_get_my_id() == 0)
      { p4_create_procgroup();
       master();
      }
   else
      slave();
   p4_wait_for_end();
   printf(" Main driver exiting normally for process %d.\n",p4_get_my_id());
}
/* constants used to define data types of messages */
#define TAGINIT 10
#define TAGDATA 20
#define TAGEND  30

/* The master will initialize all of the slaves and send the value
   of one to the first slave. The master will wait to receive a
   final value that has incremented by each slave
*/
void master()
{  int from, i, to, type, size, num_total_slaves, sum;
   int *data_array;
/* type-- used to indicate the type of message
   to -- used to indicate which slave to send to
```

P4 Code with Array

```
        size -- used to indicate the size of the message
        num_total_slaves -- used to hold the number of salve processes
        form -- used to hold id of slave to receive from
        i - used for loop control
        sum - sum of the numbers in the array
        data_array - pointer the array of numbers
    */

        char *msg;
    /* msg -- used to hold pointer to the character message */

        size = sizeof(int);
        num_total_slaves = p4_num_total_slaves();
    /* initialize all slaves */
             for (i = 1; i <=  num_total_slaves; i++)
                     p4_sendr(TAGINIT,i,(char *)&i,size);
             p4_dprintf ("\n master initialized all slaves");
    /* setup an array and give it some values */
        data_array = (int *) malloc(NUMBERS * sizeof(int));
        for( i = 0; i < NUMBERS; i++)
          data_array[i] = i;

    /* send 1 to process 1 */
        to = 1;
        msg = (char *) (data_array);
        size = MAX * sizeof(int);

    /* send the array to the other process in blocks of MAX intergers */
        for( i = 1; i <=NUMBERS; i = i +18)
          {
             p4_sendr(TAGDATA,to,(char *)msg, size);
             msg = msg + size;  /* increment the pointer to the next    */
                                /* group of MAX intergers               */
          }
             p4_dprintf("\n master sent structure to first slave");

     /* wait for reply to come from last processor */
             type = TAGDATA;
             from = num_total_slaves;

             from = -1;
     /* -1 to receive from any slave */
             p4_recv(&type, &from, &msg, &size);
             sum = *(int *)msg;
             p4_dprintf("\nThe sum of the array elements is %d\n", sum);
             p4_msg_free(msg);
          p4_dprintf("\n master exiting normally\n");
     }
```

Appendix E

P4 Code with Array

```
/*
Each slave will receive an integer from its left neighbor, add one
to it and pass it on to the next slave.
*/
void slave()
{  int next, type, sender, size, id, num_total_slaves, i, j, sum;
   int *data_array;
/* int_value -- used to store received msg as an integer
   next -- used to indicate next slave or host to send to
   type-- used to indicate the type of message
   sender -- used to indicate the id of the sending process
   size -- used to indicate the size of the message
   id -- used to hold the id of this process
   num_total_slaves -- used to hold the number of salve processes
   i j - used for loop control
   sum - sum of the numbers in the array
   data_array - pointer to the array to hold the numbers
*/
   char *msg;
/* msg -- used to hold pointer to the character message */

   typedef struct _msg
     { int int_value[MAX];
     } structure_type;
     structure_type structure;
/* structure to hold MAX integers that are sent in a group */
/* Allocate storage for the array */
   data_array = (int *)(malloc (NUMBERS * sizeof(int)));
   size = sizeof(int);
   type = TAGINIT;
   sender = 0;
/* clear out value stored in the msg */
   msg = NULL;
   id = p4_get_my_id();
   num_total_slaves = p4_num_total_slaves();
   if (id == num_total_slaves)
      next = 0;
   else
      next = id + 1;
/* Each process accepts initialization message */
   p4_recv(&type,&sender,&msg,&size);
   p4_msg_free(msg);
   p4_dprintf("\n process %d initialized\n", id);
   msg = NULL;

/* Receive an message from the left neighbor */
   sender = id - 1;
```

Appendix E

P4 Code with Array

```
   type = TAGDATA;
   for( i = 1; i <= NUMBERS; i = i + MAX)
   {
     p4_recv(&type,&sender,&msg,&size);
/* convert the received message of type char to type structure */
     structure = *(structure_type *)msg;
/* transfer this group of MAX numbers into the array of numbers */
     for(j = 0; j < MAX; j++)
       data_array[j + i] = structure.int_value[j];
     p4_msg_free(msg);
   }
   sum = 0;
/* sum the numbers and print them */
   for( i = 0; i < NUMBERS; i++)
   {
     sum = sum + data_array[i];
     p4_dprintf("\n data_array %d = %d\n", i, data_array[i]);
   }
   p4_dprintf("\nThe sum in the slave is %d\n",sum);
/* send to right neighbor */
   size = sizeof(sum);
   p4_sendr(TAGDATA,next,(char*)&sum, size);
   p4_dprintf("\n process %d terminating",id);

}
```

PROGRAMMING QUIZ
C PROGRAMMING LANGUAGE

1. Explain what the include files in C are for.
   Example: #include <stdio.h>

2. How does C declare integer and character string variables?

3. Write a **print statement in C** to print your first initial. Use the variable letter which has already been declared a character string variable to represent your initial. Use "My first initial is ".

4. Identify the labeled parts of the for loop in C below:

```
     #define number_of_random 1000


          A.          B.              C.
     for(i=0; i<number_of_random; i++)
  D. {

      E. total_num = total_num + num_array[i];

  F. }
```

Answers:

A.

B.

C.

D.

E.

F.

5. Write a comment or remark statement for the define statement in question #4 above using the remark symbols that C requires. Put your answer below.

6. You have a file named: rand.c
Write the necessary code to compile the program to an output file called rand.out.

7. Write a print statement in C to print your age. Use the variable age. Use the sentence "My age is ".

PROGRAMMING II QUIZ LINUX/UNIX

Match the following Unix commands with the correct definitions.

1. ____ cp	A. Locate a file by name and print its path.
2. ____ finger	B. Change working directory.
3. ____ kill	C. Delete a file.
4. ____ cc	D. Display a file.
5. ____ pwd	E. Status of a user.
6. ____ cd	F. Terminate a process.
7. ____ ls	G. Display names of users that are logged on.
8. ____ rm	H. Copy source file to destination file.
9. ____ cat	I. Change access mode of a file or directory.
10. ____ man	J. Compile a C program.
11. ____ ls -a	K. Display informatin about directories and files.
12. ____ mkdir	L. Display a process status.
13. ____ who	M. Print name of current working directory.
14. ____ rmdir	N. Move a file to a new name. Rename a file.
15. ____ more	O. Remove a directory.
16. ____ ps	P. List all files in current directory including hidden files.
17. ____ mv	Q. Display a file or input one screen at a time.
18. ____ find	R. Online manual/help for system commands.
19. ____ chmod	S. Displays your name as the user.
20. ____ whoami	T. Make a directory.

PARALLEL PROCESSING TEST OVER LINUX/UNIX, C, & p4

True - False Questions. Please answer with T or F.

1. _____ Researchers first started considering the idea of parallel processing in the late 1940's in order to build a computer to simulate the human brain.

2. _____ There was a lot of research in parallel processing going on in the 1960's.

3. _____ Research in parallel processing really picked up in the 1970's due to more powerful microprocessors.

4. _____ More emphasis occurred in the 1980's due to China announcing a plan to build a new generation of powerful computers that would use parallel processing.

5. _____ The United States developed a parallel computer in 1986 that had 65,536 processors.

6. _____ Parallel processing is a computer processing technique that allows many operations to be performed at the same time.

7. _____ To compile and link a P4 program for execution, you need to type: compile <filename>.

8. _____ The procgroup describes where all master processes will be executed.

9. _____ The first executable P4 statement should be: p4_initenv(&argc,argv);.

10. _____ The last executable P4 statement should be: p4_wait_for_end();

Short Answer.

11. Name the three methods for Parallel Programming.

    1.

    2.

    3.

12. Name the three <u>approaches</u> to Parallel Programs as referred to building a house and define each.

    1.

    2.

    3.

13. You have copied your file array2.c to msg1.c and are ready to <u>compile and link the p4 program for execution</u>. Write the correct command for p4 to do this.

14. What is the <u>command</u> in p4 to send messages to other processes?

15. What is the <u>command</u> in p4 to receive messages from another process?

16. What are the four arguments (pieces of information) that get sent with the process when using the send command in P4?

    1.

    2.

    3.

    4.

17. Explain what the include files in C are for.
    Example: #include <stdio.h>

18. Write a print statement in C to print your grade in this class, for example your grade is a 94.6. Use the variable grade which has already been declared. Use "My grade is " in your print statement.

19. You have a file named: random.c
Write the necessary code to compile the program to an output file called random.out.

20. Write the proper Unix/Linux code to copy lab1.c to msg1.c.

PROGRAMMING II LINUX/UNIX TEST

Match the following Unix commands with the correct definitions.

1. ____ cp	A. Locate a file by name and print its path.
2. ____ finger	B. Change working directory.
3. ____ kill	C. Delete a file.
4. ____ cc	D. Display a file.
5. ____ pwd	E. Status of a user.
6. ____ cd	F. Terminate a process.
7. ____ ls	G. Display names of users that are logged on.
8. ____ rm	H. Copy source file to destination file.
9. ____ cat	I. Change access mode of a file or directory.
10. ____ man	J. Compile a C program.
11. ____ ls -a	K. Display informatin about directories and files.
12. ____ mkdir	L. Display a process status.
13. ____ who	M. Print name of current working directory.
14. ____ rmdir	N. Move a file to a new name. Rename a file.
15. ____ more	O. Remove a directory.
16. ____ ps	P. List all files in current directory including hidden files.
17. ____ mv	Q. Display a file or input one screen at a time.
18. ____ find	R. Online manual/help for system commands.
19. ____ chmod	S. Displays your name as the user.
20. ____ whoami	T. Make a directory.

Appendix G

STUDENT SURVEY OF PROJECT

1. Evaluate human processor simulation towards your understanding of parallel processing. (addition of numbers as individuals and then together as a team).

   Excellent Good Fair Poor

2. Evaluate the lectures and demonstrations on programming in the language of C. Did this increase your understanding of C?

   Excellent Good Fair Poor

3. Evaluate the lectures and demonstrations(in library) on Linux/Unix towards your understanding of Linux/Unix.

   Excellent Good Fair Poor

4. Evaluate the lectures and demonstrations on P4 programs as done in class on the two computers with Linux as far as helping with your understanding of P4.

   Excellent Good Fair Poor

Thank you for your help and cooperation.
Mrs. Sheary

Appendix G May 18, 1995

Dear Colleague:

I too am a computer science teacher like yourself. I am presently finishing a graduate project and would appreciate your help in answering the following survey questions in order to compile some statistical information. Please take a couple of minutes to complete the survey. Enclose the completed survey in the self-addressed stamped envelope and mail it back by May 26, 1995.

1. Do you teach programming in your computer science curriculum?

   A. Yes B. No

2. How many programming languages do you teach at your school?

   A. 1 B. 2 C. 3

3. What programming languages do you teach in your computer science curriculum? Circle all that apply.

   A. Basic/QBasic B. Pascal C. Cobol D. Fortran E. C/C++ F. P4

4. Have you ever heard of parallel processing?

   A. Yes B. No

5. Do you teach P4 as a parallel processing language?

   A. Yes B. No

6. Would you be interested in learning more about parallel processing in the future?

   A. Yes B. Maybe C. No

7. Would you consider at some point in the future teaching parallel processing?

   A. Yes B. Maybe C. No

NAME:______________________________

SCHOOL:______________________________

Thank you for your cooperation. Enjoy your summer vacation!

Sincerely,

Kathy Sheary
Maroa-Forsyth H.S.
Computer Science Instructor

REFERENCES

Booch, Grady, "Coming of Age in an Object-Oriented World," IEEE Software (November, 1994):33-41.

Butler, Ralph, Ewing Lusk, "User's Guide to the P4 Parallel Programming System", Argonne National Laboratory (October, 1992 - Revised April, 1994):1-8.

Carriero, Nicholas, David Belernter, How to Write Parallel Programs - A First Course, (MIT Press, 1992) 14.

Chaudhuri, Pranay, Parallel Algorithms Design and Analysis, (Prentice Hall, 1992) 4-5.

Davidson, Rob, "Multitasking vs. Multiprocessing In Industrial Control Applications," Chilton I and CS, 66 (February 1, 1993): 25-28.

Gehringer, Edward F., "Parallel Processing", Grolier Electronic Publishing, Inc., 1994.

Hord, R. Michael, Parallel Supercomputing in MIMD Architectures, (CRC Press, 1993) 17.

Kumar, Vipin, Ananth Grama, Anshul Gupta, and George Karypis, Introduction to Parallel Computing, (The Benjamin Cummings Publishing Company, Inc. 1994) 2-3.

Lafferty, Edward L., Parallel Computing - An Introduction, (Noyes Data Corporation, 1993) 34-61.

Lazou, Christopher, Supercomputers and Their Use, (Oxford University Press, 1986) 1-2.

Miller, Russ, "The Status of Parallel Processing Education", IEEE Computer, (August, 1994): 40-43.

Narad, Chuck, "Multiprocessing to Lead The Way", Electronic Design, 41 (November 22, 1993): 52.

Nevison, Christopher H., "Parallel Computing for Undergraduates", National Science Foundation and Colgate University, (June, 1994): 4.

Ramakrishna, B., and Joseph Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective", The Journal of Supercomputing, 7 (May 1, 1993): 9-50.

Schmit, Julie, "Developer: Key to Computers is Brain", USA Today, August 17, 1994: 2B.

Welsh, Matt and Lar Kaufman, Running Linux, (O'Reilly and Associates, Inc., 1995) 1-7.

Kathryn A. Sheary

Education	B.S. in Education, Eastern Illinois University	
Experience	Jr. High Mathematics, Piasa Southwestern Jr. High	
	Elementary Education, Breese Elementary School	
	Jr. High Mathematics, Robinson Nuttall Middle School	
	High School Computer Science and Jr. High Mathematics, Maroa-Forsyth High School	
	System Operator, Maroa-Forsyth High School	
Course Work	Spring 92	PAC 414 - Science, Technology, and Public Policy
	Spring 94	CSC 474 - Intro to System Programming and Operating Systems
	Spring 94	CSC 476 - Intro to Microprocessors and Computer Architecture
	Summer 94	CSC 484 - Intro to Parallel Processing
	Fall 94	CSC 478 - Intro to Software Engineering
	Fall 94	CSC 574 - Operating Systems
	Spring 95	CSC 578 - Software Engineering
	Spring 95	CSC 549 - Master's Closure Project