CSE@UTA Technical Report
Research Experiences for Undergraduates in Dynamic Distributed Real-Time Systems Program

Summer 2000

by
Jeff Marquis
Chris Forrest
Kshiti Desai
Geoff Dale

This report supported in part under National Science Foundation, Contract number EIA-9820440, Research Experiences for Undergraduates in Dynamic Distributed Real-Time Systems.

Introduction

The National Science Foundation (NSF) Research Experiences for Undergraduates (REU) in Dynamic Distributed Real-Time Systems Program for Summer 2000 (REU 2000) in the Computer Science and Engineering Department at The University of Texas at Arlington (CSE@UTA) was designed to give three sponsored undergraduate students theoretical knowledge, exposure to advanced programming tools, and participation with applied research in parallel and distributed processing. This was achieved through classroom instruction, hands-on application development, introduction to the state-of-the-art PARSA^{TM} Software Development Environment and the ThreadMan^{TM} Thread Manager, and exposure to applied research being conducted to advance the capabilities of PARSA to support distributed cluster computers. We believe the students have benefited from this program by combining the theoretical basis and hands-on exposure to software development issues in parallel and distributed processing.

Classroom Instruction: The REU 2000 students were required to register for Introduction to Parallel Processing, CSE4351 (http://reptar.uta.edu/NOTES4351/cse4351.html), taught by Dr. Bob Weems, CSE@UTA Professor. In the course the students were introduced to parallel programming concepts and were assigned a series of practical programming assignments using multi-threading for shared memory systems and message passing for distributed memory systems. Specifically, the students were introduced to the pthreads library for exploiting parallel processing on shared memory systems and the Message Passing Interface (MPI) for exploiting parallel processing on distributed memory systems. The students acquired the theoretical knowledge and hands-on parallel programming skills through classroom instruction that allowed them to participate in and materially contribute to the REU 2000 program.

Introduction to PARSA: The PARSA^{TM} Software Development Environment and ThreadMan^{TM} Thread Manager are commercial software products developed at CSE@UTA under license to Prism Parallel Technologies, Inc. (www.prismpti.com). The current commercial versions of PARSA and ThreadMan are used for developing parallel multi-threaded software for shared memory systems. CSE@UTA and Prism Parallel Technologies continue to perform collaborative fundamental and applied research to advance the capabilities of PARSA and ThreadMan. The REU 2000 students were given exposure to these on-going research and development activities, which applied the theoretical knowledge and parallel programming skills obtained through classroom instruction.

The REU 2000 students were tasked with replicating applications developed in CSE4351 using PARSA and ThreadMan. At first, students were simply given access to PARSA and the PARSA Programming and Reference Manual and were instructed to figure out the most efficient way to develop applications in PARSA. This part of the program forced the students to independently learn to use state-of-the-art programming tools with a minimum of instruction, which we believe will serve them well throughout their computer science careers. During this phase of the program the students were given cursory guidance by Jeff Marquis, a CSE@UTA Faculty Research Associate and the founder of Prism Parallel Technologies, on the use of PARSA and ThreadMan.

The students successfully replicated several of the class assignments in PARSA, which exposed them to the benefits that PARSA, ThreadMan and advanced programming tools provide. Namely, the students learned i.) that PARSA reduces the level of expertise required by programmers by abstracting programmers from the low-level details of multi-threaded programming; ii.) that PARSA significantly reduces the amount of code that must be generated by an automated code generation process that generates all multi-threading directives and data passing code for shared memory systems, iii.) that ThreadMan efficiently manages the execution of multi-threaded software at run-time relieving programmers of this tedious and difficult task, and iv.) that PARSA and ThreadMan produce very efficient and scalable code. The students were then given formal training on the use of PARSA and asked to redevelop class assignments in PARSA, which exposed them to the benefits of rigorous and formal training.

Applied Research: To provide the REU 2000 students with exposure to applied research the students were instructed to work with a CSE@UTA graduate student, Mr. Kannan Bhoopathy, who was performing fundamental and applied research that extended PARSA and ThreadMan to support the development of applications for distributed cluster computers, an emerging architecture popularized by networked clusters of symmetric multi-processors (SMPs), the Compaq SC, IBM SP3, and other contemporary supercomputer class architectures. The students were assigned the task of developing applications for distributed clusters following Mr. Bhoopathy's methods, validating the methods proposed by Mr. Bhoopathy, and doing a performance analysis of the results.

By working closely with Mr. Bhoopathy the REU 2000 students were exposed to many facets of performing applied research, providing them with a rewarding experience. Specifically, the students were forced to understand the new method for supporting distributed cluster computers devised by Mr. Bhoopathy. This was achieved by successfully developing distributed cluster computer applications using the enhanced methods integrated into PARSA by Mr. Bhoopathy. The REU 2000 student's independent work validated Mr. Bhoopathy's work while exposing the students to the rigors of performing research. To summarize their results the REU 2000 students performed a detailed performance analysis on the applications developed for distributed cluster computers.

REU 2000 Summary: The REU 2000 students worked as individuals and as a team throughout the summer. The individual work exposed each to the rigors of performing research independently, which will serve them well throughout their computer science careers. The group work forced the students to collectively solve larger scale problems by partitioning larger scale problems amongst themselves in such a way that the work would be completed and that each student would perform approximately the same level of work. We believe the REU 2000 students benefited immensely from the REU 2000 program.

REU 2000 Technical Report Organization: In this report the REU 2000 students present an overview of their experiences and the lessons learned during this program. We at CSE@UTA are very pleased with the performance of the students, and we believe they will benefit from their work on this year's program. For additional information on the REU program at CSE@UTA visit our web site at http://cygnus.uta.edu/reu/ or contact Dr. Behrooz Shirazi, Chairperson and Professor at CSE@UTA, at shirazi@cse.uta.edu.

The first section provides an overview of the classroom instruction that provided the underlying knowledge needed by the students to participate in the REU 2000 program. The first section is written by Mr. Geoff Dale, who will return to finish his senior year at Texas Christian University in the fall of 2000. CSE@UTA has committed to continue supporting Mr. Dale during the fall 2000 semester with REU funds.

The second section provides an overview of the PARSA Software Development Environment and ThreadMan Thread Manager. The second section is written by Ms. Kshiti Desai, a senior in computer science and engineering at CSE@UTA. CSE@UTA has committed to continue supporting Ms. Desai during the fall 2000 semester with REU funds.

The third section reports on the applied research performed by the REU 2000 students on developing applications for distributed cluster computers. The third section is written by Mr. Chris Forrest, who is graduating from CSE@UTA in August 2000. We are happy to report that Mr. Forrest has decided to enter the graduate program at CSE@UTA, in part because of his experience with the REU program.

Classroom Instruction

As part of our research experience, the members of the research program were required to take the parallel processing class taught by Professor Bob Weems at the University of Texas at Arlington while also working on the research project. The purpose of taking the class was to introduce the various programming techniques and the concepts involved with parallel programming. This knowledge would be used to reinforce the future research involved in the research program.

The main objective of the class was an introduction to the variety of topics necessary for developing parallel applications. Some of the goals of the class included the ability to implement small applications on shared-memory multiprocessor (Linux SMP) using pthreads and in message-passing paradigm using MPI, and the understanding of concepts of parallel algorithms, elementary topologies, and compiler concurrentization concepts. The class semester was structured in two parts. Each part of the semester involved discussion and note taking over various topics in parallel programming with two labs to reinforce the covered topics and one test at the end of each portion of the semester, covering all of the topics covered prior to the test. The C programming language was used to demonstrate the majority of the topics covered in the class.

The class began with an introduction to parallel programming using Linux Threads (pthreads). The class learned how to create parallel code with multiple processes that would work together to execute code in parallel. Pthreads are used on shared memory systems, meaning every process has access to the same memory locations. The class discussed the various methods of coding pthreads and process management, including static and dynamic scheduling. The two forms of static scheduling are the interleaving form, or horizontal spreading, and the contiguous form, or vertical spreading. Interleaving is the simplest form of static scheduling, while the contiguous form creates a simple mapping for the execution of the code. Dynamic scheduling, or self-scheduling, allows threads to be allocated on the run to take available work. The class went over various examples involving these three scheduling methods, including parallel versions of a summing program, shell and quick sort, and Warshall's algorithm. Pthreads and the different scheduling techniques involved with them were used to introduce parallel programming to the research group.

The class then began to study parallel processing through the use of the Message-Passing Interface, or MPI. The class had to focus on how processes could execute code in parallel without having a shared memory between processes. Message passing and the different techniques involved with message passing were studied along with the various functions that are needed to allow MPI to execute correctly. Several examples of programs using MPI were discussed in class, including Warshall's algorithm, a hashing program, and table balancing. The class was introduced to parallel programming techniques on machines that did not share memory through the use of MPI.

Concepts important to parallel processing where next covered in the class. The topic of synchronization techniques for both shared memory and message passing programs was one of these concepts discussed. Synchronization prevents race conditions and bottlenecks from occurring in the code and ensures that the code produces the correct output. For shared memory programs, the concepts of locks, semaphores, and barriers 3 were discussed. The class also reviewed the four forms of message passing, which are the remote procedure call, dynamic process creation, asynchronous message, and the rendezvous.

The class went on to discuss shared memory models for creating efficient algorithms. The PRAM model was discussed here. Topics such as list ranking, Euler trees, and various tree traversals were also covered. The class also discussed speed-up and efficiency analysis and how this analysis applies to parallel programming.

Routing techniques and interconnection networks were discussed by the class. Topologies involving linear arrays, meshes, hypercubes, and toruses were covered. Techniques of routing through various networks were discussed. Some of these techniques included the Benes network, the butterfly technique, and the shuffle exchange. Examples of all of these topics and techniques were discussed in class.

Using parallel processing to do numerical problems was another topic covered in the class. Solving systems of linear equations on both shared memory and distributed memory systems by using elimination methods, including Gaussian, LU, and the householder method, and by using iteration were discussed and enforced through examples. The communication involved in performing matrix multiplication and matrix transpose was also covered in the class.

Other topics included sorting techniques that can be applied in parallel processing, including linear arrays and the bitonic mergesort. The ranking and unranking of combinatorial objects was discussed. Lexicographic ordering concepts and numbering techniques in parallel were also reviewed.

Along with two tests, the class had four lab assignments that reinforced the material that had been covered in class. The first lab involved modifying a sequential program to work in parallel with two threads using pthreads. A speed-up and efficiency evaluation based on the results of the program was to be taken after the completion of the program. The results of this lab demonstrated the increase in speed-up and efficiency that can be gained by running a program in parallel compared to running a program sequentially.

The second lab assignment used MPI techniques on a client/server hashing program. This lab involved modifying a MPI program where every process acted as either a client or a server into a MPI program where every process acted as both a client and a server. This lab served as a basis for programming in parallel using MPI techniques and demonstrated the effectiveness of writing efficient MPI code.

The third and fourth lab assignments both involved creating a parallel version of Lukes' technique to determine the bisection width of an undirected graph. A pthread version using two threads and a MPI version using various numbers of processors on different machines was required. These lab assignments provided an experience with parallelizing a non-trivial irregularly structured application.

The parallel processing class provided a foundation for the work that the research group would use throughout the summer. The class provided an introduction to parallel processing, the concepts involved with parallel processing, the types of problems parallel processing can be used to solve quickly and efficiently, and the techniques needed to write parallel code. The research group used the knowledge gained from the class as the building block for the rest of the research it would perform throughout the summer.

Introduction to PARSA and ThreadMan

As we learned software development methods in our classroom instruction we were tasked with learning the PARSA software development methodology. PARSA supports a graphical programming methodology specifically designed for developing parallel software. Using PARSA's graphical programming methodology software developers are abstracted from the low level details of developing multithreaded software. Specifically, PARSA does not require programmers to develop multithreaded directives, pass data between threads, or manage and coordinate the execution of threads at run-time. This saves a lot of time by making it as easy to develop multithreaded software in PARSA as it is to develop sequential software. It also reduces the complexity of developing parallel software as we learned after using PARSA for the first time.

PARSA applications are made up of graphical objects (or GOs) and arcs. Each graphical object simply represents an application task to be performed. Graphical objects consist of an interface section and a functionality section. The interface section describes the interface that each graphical object will have with other graphical objects within an application. The functionality section is programmed by defining the INPUT and OUTPUT variables that the graphical object is dependent on for execution and will generate after execution, respectively. You simply make C variable declarations to define INPUT and OUTPUT variables. The graphical object ends up with one INPUT port and one OUTPUT port for each INPUT and OUTPUT variable declared in the interface section. The functionality section is where the graphical object's task is programmed in sequential C without any threading directives.

PARSA supports 3 different types of graphical objects: user-defined, forall and while. User-defined graphical objects are just tasks and are programmed like functions in C. Forall graphical objects provide support for regular parallelism, which gives good opportunity to get good speedups. While graphical objects provide support for repeat parallelism, which improves performance by exploiting parallelism within the body of a convergent loop.

Arcs are used to connect graphical objects together. Arcs represent the data and control flow between graphical objects, and it simply connects OUTPUT ports of source graphical objects with INPUT ports of destination graphical objects.

Once an application is complete PARSA Processing converts the graphical representation of an application into multithreaded source code. The source code generated by PARSA includes the code programmed into the functionality section of each graphical object plus all necessary threading directives and code for passing data between the graphical objects. The source code is compiled and linked with the ThreadMan Thread Manager, which manages the execution of the application at run-time.

Our first exposure to PARSA was just access to the tool and the PARSA Programming and Reference Manual. We redid the first class assignment in PARSA, which was to determine the Least Common String from a set of 45402 words in a dictionary. In class we had to develop the application by hand using pthreads. As part of our research we redid the application in PARSA. Figure 1 shows the Least Common String in PARSA.

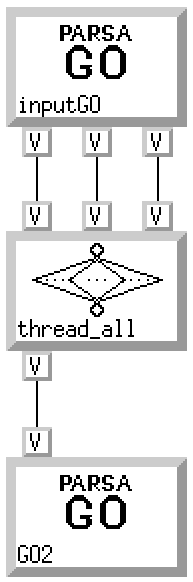


Figure 1. Least Common String application in PARSA.

The first user-defined GO, called inputGO, allocates space for the required data structures and reads in the information from the dictionary. The dictionary data is passed to the forall GO, called thread_all. thread_all is programmed to split the work into two threads and the result is passed to GO2. GO2 prints the results.

The first thing we noticed was how easy it was to develop multithreaded software with PARSA. We didn't have to even know we were developing multithreaded software because PARSA developed the entire multithreaded source and the data passing code for us. Also, we didn't have to figure out how to manage the threads when they ran because ThreadMan did that for us. So PARSA proved to be a very good tool for making programming multithreaded software easy.

We did a performance analysis between the hand code developed in class and the software developed with PARSA. The performance results are shown in Table 1. As you can see the performance of the hand-coded and PARSA implementations are comparable.

Programs	**Timings (CPU time)**
Sequential Version	1.04
Hand-coded pthreads Version	0.30
PARSA implemented Version	0.33

Table 1. Performance comparison between hand-coded and PARSA generated multithreaded software.

Table 2 demonstrates one of the primary benefits of PARSA. The hand-coded implementation required us to develop each line of source code ourselves and was 311 lines of code. To develop the application in PARSA only required 63 lines of code. This produces significant savings in the time it takes to develop multithreaded software using PARSA.

	Number of Lines
Hand-coded.	311
Developed in PARSA	63
PARSA Generated Code	558

Table 2. Lines of code count comparison between hand-coded, developed in PARSA, and automatically generated by PARSA.

After we learned the basics of PARSA on our own we got trained in using PARSA by Mr. Marquis. The training gave us a better understanding of how to use PARSA and all the features of PARSA. It was very helpful to see the training and it made us understand PARSA better. It also showed us some innovative ways that we could use PARSA to develop multithreaded functions and eCompute applications.

Applied Research

Introduction: One of the goals of the REU team was to help perform applied research on the PARSA and ThreadMan technology being jointly developed by CSE@UTA and Prism Parallel Technologies. PARSA is a programmer's development tool aiding in writing multithreaded applications. By using graphical representations of data flow, the application designer does not have to worry about the low-level thread details. A logical advancement of this software tool is the ability to assist programmers with distributed applications that could span across multiple machines as well as multiple processors per machine. We were asked to assess the feasibility of an approach devised by Mr. Kannan Bhoopathy that incorporates the Message Passing Interface (MPI) using the PARSA software development methodology. Topics that we investigated were:

1. Determine the feasibility and difficulty of hand coding MPI calls into the code generated by PARSA to support distributed cluster computers.
2. Obtain timing results of sample applications to determine what performance benefits may result using distributed cluster computers.
3. Determine if any problems exist that might hinder MPI code to be automatically generated by PARSA, which would relieve software developers of this tedious and time-consuming task.
4. Review a requirements document generated by Mr. Bhoopathy for PARSA to automatically generate MPI code to support distributed cluster computers.

Sample Application Selection: When searching for applications to code, we decided to look for functions that were typically done in parallel. After talking with our sponsor, we decided to program two applications. One program would use matrix multiplication and the other would use merge sort. Both matrix multiplication and merge sort are classic parallel applications. These applications were also picked because of their high use of the CPU (when given large inputs). Another reason these applications were picked was because they can benefit from parallelism, which would help recognize how efficient the code generated would be.

We worked closely with Kannan Bhoopathy who devised a scheme for using the PARSA graphical programming methodology to support distributed cluster computing. His work provided the basis of our work because we were validating and assessing the performance of the scheme he devised.

Merge Sort: The merge sort program was divided into 4 different sections as shown in Figure 2.

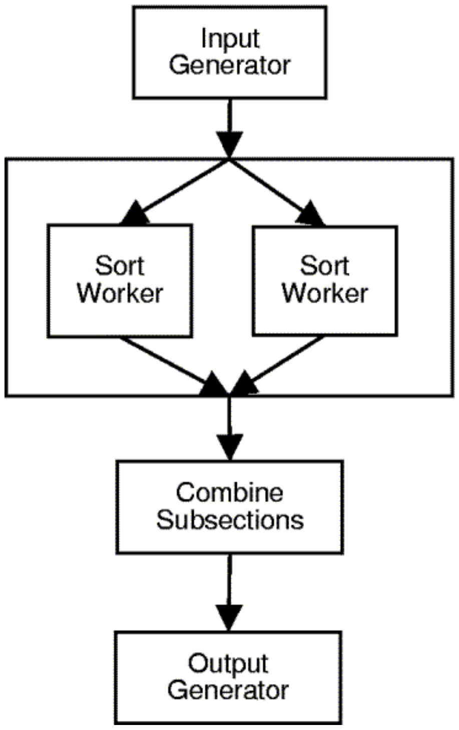
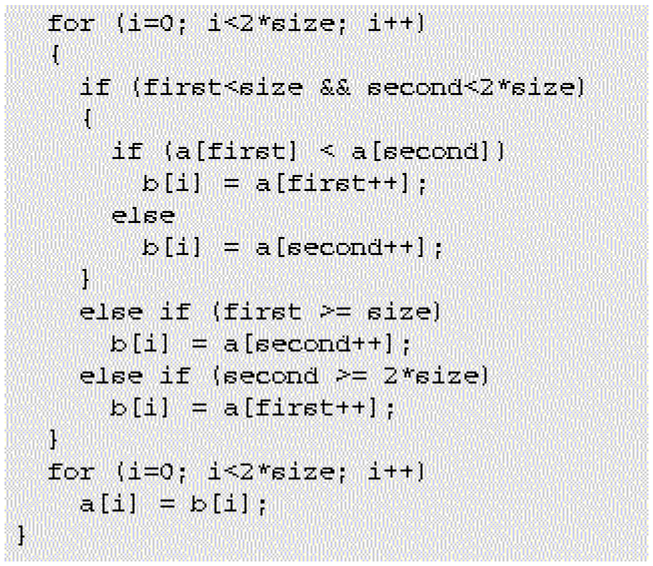


Figure 2. Merge Sorting

The input generator generates random integer data (with a specific random seed for repeatability) that is placed into an array. This array is then subdivided to worker threads that sort the information relative to their smaller arrays. The smaller arrays are then combined together to generate the final output. We wanted to see what the penalty would be for placing the sorting task into one process. In order to accomplish this, we used the above concept to generate the following applications:

1. A simplistic merge sort application that executes sequentially (i.e., as a single thread).



```
  for (i=0; i<2*size; i++)
  {
    if (first<size && second<2*size)
    {
      if (a[first] < a[second])
        b[i] = a[first++];
      else
        b[i] = a[second++];
    }
    else if (first >= size)
      b[i] = a[second++];
    else if (second >= 2*size)
      b[i] = a[first++];
  }
  for (i=0; i<2*size; i++)
    a[i] = b[i];
}
```

Figure 3. Merge sort as a single thread.

   The above single thread application creates an array double the size of the original array (notated in the code as b). The newly created array is then used to partition the data into two different sections. Data is placed in either section based on comparing the elements within the original array. After the data has been separated and sorted, the information is brought back together to produce the resultant output.

2. A merge sort application designed in PARSA that uses a forall graphical object and a while graphical object. The forall graphical object creates multiple threads that allow independent iterations of the function to run in parallel. The forall graphical object was used because the work performed by the loop iterations is independent, and therefore, can most benefit using a forall graphical object. This version also used a while graphical object to merge the sorted sublists together. A while graphical object was used because the merging task exhibits repeat parallelism, and while graphical objects exploit the run-time benefits of repeat parallelism.

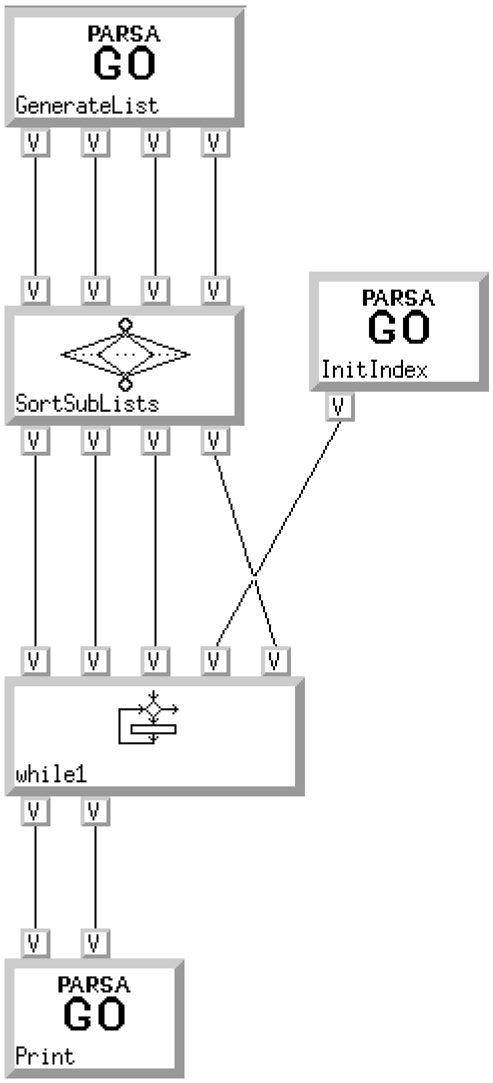


Figure 4. Merge sort in PARSA

The PARSA application performs the sorting operation in four different stages. Once the randomly generated data (with a specific random seed for repeatability) is created in the “GenerateList” graphical object, the array (and other required information) is passed to the “SortSubLists” forall graphical object. SortSubLists creates multiple (independent) worker threads and allows the threads to work on a different region of the input array. The generated results are passed to the “while1” while graphical object, which merges the sorted sublists. The information is then passed to the “Print” graphical which prints the sorted and merged list.

3. Merge sort using the PARSA programming methodology with MPI support, or MPI-PARSA. The MPI-PARSA version of merge sort has the same overall layout as shown in Figure 4, but the graphical objects contain calls to MPI routines for passing data between graphical objects. The required MPI routine calls were hand coded into the graphical objects. With the MPI code embedded into the application the PARSA programming methodology is maintained, but the graphical objects execute as MPI processes, not as threads. For merge sort five MPI processes are created, one for each graphical object. Notice that each process can itself spawn multiple threads, which supports distributed cluster computing. For example, forall graphical object SortSubLists creates multiple threads to perform the work.

Timing Analysis: After developing the different versions of merge sort we performed a timing analysis to compare and contrast the various methods used to develop the application. We used a two-processor Pentium II (named zig) to collect timing results. We expected that the PARSA-MPI version would be slower than the multithreaded version generated by PARSA because of the overhead associated with passing the data between MPI processes and the lack of parallelism between the graphical objects of merge sort (see Figure 4).

The work was divided as shown in Table 3:

	Serial	**PARSA only**	**PARSA-MPI**
Machines used	Zig	Zig	Zig
Memory size of machine	127MB	127MB	127MB
Swap size of machine	72MB	72MB	72MB
Number of processors working collectively	1	2	2
Number of work threads per machine	1	2	2
Number of threads combining the result array	Not applicable	1	1
Number of active MPI processes	0	Not applicable	5

Table 3. Work divided between systems and the system configurations.

After running the different versions of merge sort with varying sized arrays the following timing data was obtained (all times are in seconds):

	10000	**30000**	**50000**	**70000**	**100000**
Serial	4	33	90	177	361
PARSA	2	17	46	90	182
PARSA-MPI	12	20	57	93	191

Table 4. Timing data collected for multiple runs of merge sort.

The results show that the MPI version is slower than the PARSA version as expected. This is due to the overhead of passing messages between MPI processes and the lack of parallelism between graphical objects (i.e., MPI processes). For example, a performance gain could be obtained if two forall graphical objects were used to sort the sublists where each forall graphical object would be executed on a different system. The effectiveness of this method is demonstrated in the matrix multiplication application presented below.

Matrix Multiplication: The matrix multiplication program was divided into 5 different sections as shown in Figure 5.

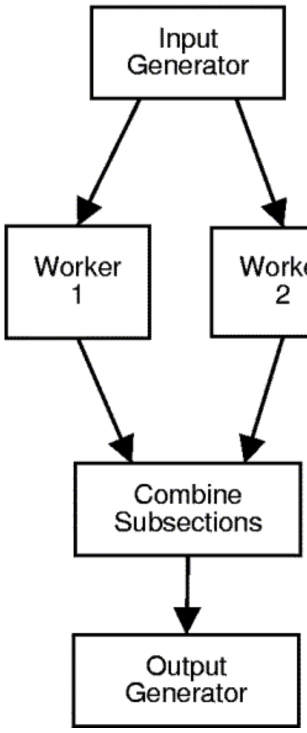


Figure 5. Matrix Multiplication.

The input generator generates random integer data (with a specific random seed for repeatability) to be placed into two arrays, A and B. A third array, C, is also created to store the result. The C array is passed to the worker sections for storing the results of the multiplication. Instead of having a single section generate the full result array, the worker sections split the work between them evenly (one section multiplies the upper-half of the array while the other section multiplies the lower-half). We also developed another variation of matrix multiplication that used four workers to generate the result. We generated the following versions of matrix multiplication:

1. A sequential version of matrix multiplier application (i.e., that executes as a single thread).

```
for (i=0; i < arraySize; i++) {
    for (j=0; j < arraySize; j++) {
        C[i][j] = 0;
        for (k=0; k < arraySize; k++) {
            C[i][j] = C[i][j] + A[i][k]*B[k][j]
        }
    }
}
```

Figure 6. Matrix multiplication as a single thread.

The single thread application is based on the idea of multiplying a row of A by columns within B. This is done all within a nesting of *for* loops.

2. Matrix multiplication designed in PARSA that uses forall graphical objects. Each forall graphical object creates multiple independent threads that perform the matrix multiplication task in parallel. The forall graphical object could be used because instantiations of the loop iterations in matrix multiplication are independent of all instantiations. This version also has a forall graphical object that merges the two data regions together. A forall graphical object could be used because merging the two arrays can be done safely in parallel.

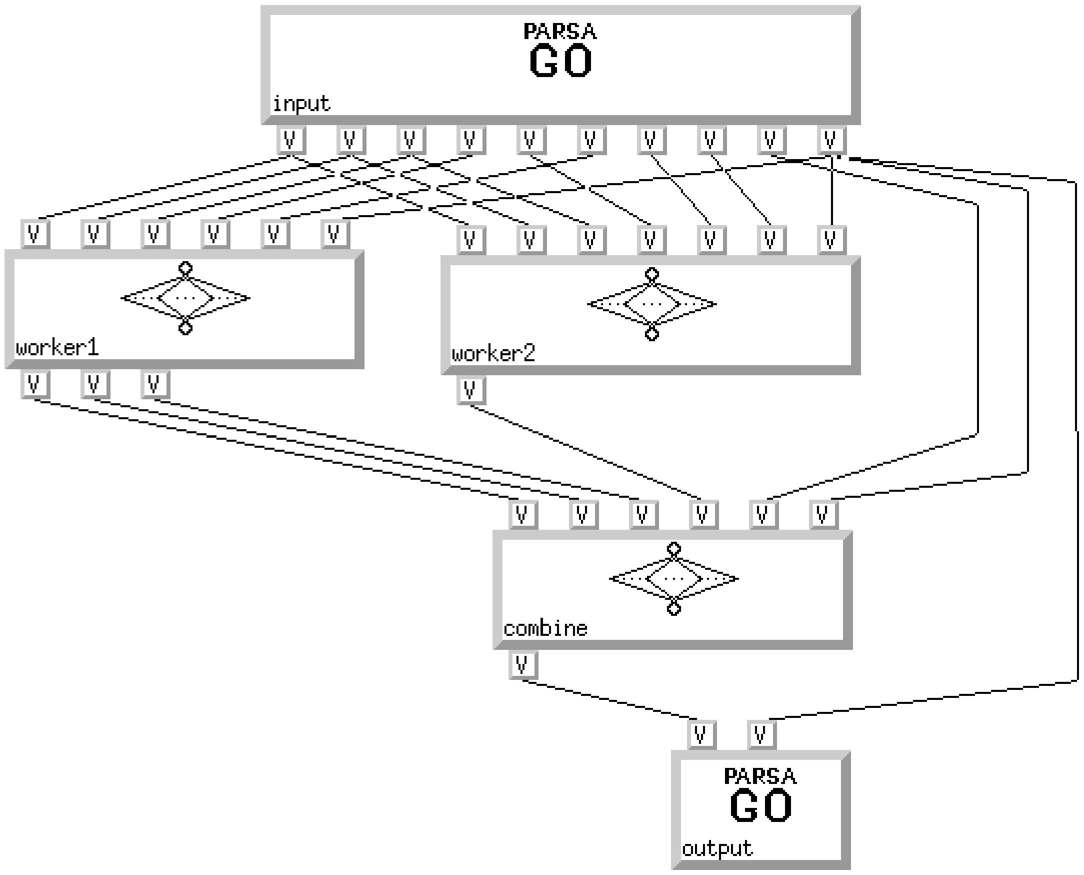


Figure 7. Matrix multiplication defined within PARSA

The PARSA version of matrix multiplication looks complicated, but it simply follows the design shown in Figure 5. The “input” graphical objects creates the A, B, and C arrays. Then randomly generates data (with a specific random seed for repeatability) for the A and B arrays. The arrays are then passed to the two worker forall graphical objects, “worker1” and “worker2”. Each worker forall graphical object creates a user-defined number of threads to perform the multiplications. Since the total work is divided between the two worker graphical objects, each will calculate an independent region of the output array C. Once the subregions of array C have been calculated the “combine” forall graphical object combines the two subregions together into C, which is then passed to the user-defined graphical object “output”. Graphical object output either displays the results or stores the results in a file.

3. A matrix multiplier application designed using PARSA based on the previous implementation with MPI support. As with the PARSA-MPI version of merge sort, the PARSA-MPI version of matrix multiplication has the same layout as Figure 7, but each graphical object has hand-coded calls to MPI routines embedded within them. Again, graphical objects in PARSA-MPI applications are each an MPI process, not a thread. Therefore, five processes are created for matrix multiplication. It should be noted that each forall graphical object in matrix multiplication can itself generate multiple threads that can execute in parallel. Hence, the PARSA-MPI version of matrix multiplication has multiple MPI processes that can execute concurrently on different machines, and each MPI process can execute in parallel. This demonstrates how the PARSA programming methodology can be used to support both multithreading and message passing for distributed cluster computers.

Timing Analysis: After developing the different versions of matrix multiplication we performed a timing analysis to compare and contrast the various methods used to develop the application. We had three different multiprocessor machines available for testing which proved to be very helpful for comparing the different implementations. We used a two-processor Pentium II (named zig), a second two-processor Pentium machine (named mustard), and a third two-processor Pentium machine (named ketchup).

The work was then divided amongst the different systems as shown in Table 5:

	Serial	**PARSA only**	**PARSA-MPI / Load Balanced**	**PARSA-MPI / Not Load Balanced**
Machines used	Zig	Zig	Mustard, ketchup	Mustard, ketchup
Memory size of machine	127MB	127MB	Both machines have 256MB	Both machines have 256MB
Swap size of machine	72MB	72MB	Both machines have 260MB	Both machine have 260MB
Total number of processors working collectively	1	2	4 – This is done via MPI	4 – This is done via MPI
Number of work threads per machine	1	2	2	2
Number of threads combining the result array	Not applicable	Not applicable	2 threads running on Ketchup	2 threads running on Ketchup
Number of active MPI processes	0	0	2 Mustard 3 Ketchup	1 Mustard 4 Ketchup

Table 5. Work divided between systems and the system configurations.

Matrix multiplication was developed so the user could specify the input array size. We chose the following array sizes for performing our timing analysis: 100x100, 500x500, 1000x1000, 1500x1500, 2000x2000, 2500x2500, and 3000x3000.

After running the different versions of matrix multiplication the following timing data was obtained (all times are in seconds):

	100x100	**500x500**	**1000x1000**	**1500x1500**	**2000x2000**	**2500x2500**	**3000x3000**
Serial	0.23	34.47	282.64	970.64	2360.6	4976	8473
Parsa Only	0.13	15.88	126.12	312.53	1058.93	2160	3840.48
Parsa MPI Load Balanced	24.76	29.85	115.59	324.2	712.55	1384	2438
Parsa MPI Non Balanced	25.36	27.14	104.09	299.95	685.2	1342	2383

Table 6. Timing data collected for multiple runs of matrix multiplication.

The reason for testing the PARSA-MPI applications as both balanced and non-balanced is due to memory issues. By equally dividing the number of PARSA-MPI processes, the memory consumption was equally distributed. The non-balanced version suffered a performance penalty continually running out of memory because it had to use the system's swap space (even with 256MB of memory!). The same was true of the PARSA Only version. The memory issues are discussed below.

Findings Using Parsa-MPI

We discovered several issues when executing the various versions of the applications.

Issue 1 – Memory Consumption
Memory appears to be a problem when larger input data sets are used (e.g., matrix multiplication with input arrays of size 3000x3000). This may seem to be a simple matter of freeing unused arrays throughout the life of the test application. However, we found the problem to be much worse. Malloc and free can give back memory to the operating system, but there are certain circumstances when this is not possible. One might wonder how 250MB of memory can become consumed by a small test application. We found that memory is consumed only when the (large) arrays are passed via MPI and dynamically allocated memory is not released back to the operating system.

Issue 2 – Passing Dynamic Memory
We discovered that passing non-primitive data types (such as dynamically allocated arrays, structures, linked lists) between distributed clusters causes problems. In PARSA v1.2 passing pointers to non-primitive data types is allowed because PARSA exploits the shared memory model of multithreaded systems. However, passing pointers on distributed memory systems and distributed clusters this model breaks down because a pointer is a memory address, which has no meaning when passed to another autonomous system in the distributed architecture.

Issue 3 – Passing Structures in Heterogeneous Environments
Another issue that comes with distributed computing is the non-uniformity of machine types that you wish to work together. MPI can allow for primitive data type conversion. However, MPI must be informed of other structures before they can be used. This requires additional steps to be developed to handle these types of situation where data types may differ between different processors in the system (e.g., little endian versus big endian).

Issue 4 – Non-Standard C Functionality in PARSA-MPI
There are certain functions that must not be used in PARSA-MPI. For example, the C exit function is acceptable to use within PARSA since when a thread exits you usually desire the whole program to exit. This is not the case under MPI where each section is an actual process. If a section of an MPI application exits it may cause the application to deadlock. MPI has a directive to inform the application to quit (MPI_Abort), however this is not standard C.

Conclusion

In conclusion, the REU 2000 program was well conceived and very informative. It provided us with not only theoretical knowledge, but practical, hands-on research experience, which we each think will benefit us throughout our computer science careers. We want to thank the National Science Foundation for their support of this program, and we would encourage them to continue and expand such programs. We think more students could benefit from this program.

We would also like to thank the faculty, staff and students in CSE@UTA for their support and encouragement. Specifically, we would like to thank Kannan Bhoopathy, Jeff Marquis, Dr. Bob Weems, Dr. Diane Cook and Dr. Behrooz Shirazi.

Chris Forrest
Kshiti Desai
Geoff Dale