Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment

JAMES C. BROWNE, TAEJAE LEE, AND JOHN WERTH, MEMBER, IEEE

Abstract—This paper reports on the initial experimental evaluation of the Reusability-Oriented Parallel Programming Environment (ROPE), a software component reuse system developed at the University of Texas at Austin. ROPE helps the designer find and understand components by using a new classification method called structured relational classification. ROPE is a part of a development environment for parallel programs which uses a declarative/hierarchical graphical programming interface. This interface allows use of components with different levels of abstraction, ranging from design units to actual code modules. ROPE supports reuse of all of the component types defined in the development environment.

The experimental design included metrics such as fraction of code in a program consisting of reused components, development time and error rates, and qualitative metrics such as user satisfaction and user perceived utility. The subjects for the experiments were undergraduate and graduate students in the Departments of Computer Sciences and Electrical and Computer Engineering.

The development time was drastically cut for all programs and the average fraction of code in a given program which was reused from the ROPE library was about 80 percent. Programs developed with the aid of ROPE were found to have error rates far less than those developed without ROPE. These results, while gratifying, are not surprising. They show that ROPE attains a high level of reuse for programs of modest size and complexity. These experiments are the necessary first step in a systematic evaluation of reuse in the CODE/ROPE environment.

Index Terms—Experimental evaluation, parallel programming, programming environments, software reuse.

I. PROBLEM STATEMENT, OVERVIEW, AND PREVIOUS WORK

A. Problem Statement

ENVIRONMENT environments which effectively promote reuse of software components are a dominant concern of software engineering. There have been a plethora of concepts proposed for promotion of reuse, and more than a handful of these concepts have been realized in some implementation of a software development environment. There have, however, been only a handful of systematic and in-depth experimental evaluations of the effectiveness of implementations of these concepts in attaining the goals of reuse. This is unfortunate because, as was said by Conte et al. [8],

"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind."

Progress in reuse of software modules may come without experimental evaluation of concepts and implementations, but if so it will be because a self-evident and obvious truth suddenly appears. Such events are of unfortunate rarity in software engineering. It was not obvious to us that the concepts we espouse fell into the category of self-evident truth. We have, therefore, felt it important to subject them to experimental evaluation.

B. Overview

This paper describes the first phase of the experimental evaluation of the ROPE (Reusability-Oriented Parallel Programming Environment) reusability system developed at the University of Texas. The system is part of a development environment, called CODE (Computation-Oriented Display Environment), whose purpose is the construction of parallel programs using a declarative and hierarchical graph model of computation. ROPE features reuse of both design components and code components; this is made possible by the fact that CODE uses the same language at every level of abstraction. ROPE focuses on the key issues of reusability; finding components, then understanding, modifying, and combining them.

The paper reports experimental investigations of user productivity and software quality for the CODE programming environment, with and without the ROPE reusability system, and measures of user satisfaction and user perceived utility for the reusability system. The experimental population included undergraduate and graduate students from computer science and computer engineering. We have attempted to design and conduct a controlled experiment evaluating the effectiveness of both the concept base and the engineering decisions made in the construction of ROPE reusability environment for parallel programming.

The experiments have demonstrated very high degrees of reuse and sharp decreases in development time for programs of modest size. These results are not surprising but are encouraging and are the necessary first step towards
an in-depth evaluation spanning large application systems and evaluation of more subtle metrics such as the relative effectiveness of the several models of classification available in ROPE.

C. Related Research

A review of the literature on programming environments in general, and reusability systems in particular, reveals that most of them have not been subjected to any experimental evaluation. The following briefly describes the conclusions from those systems for which experimental evaluations have been published. Lenz et al. [16] report on productivity and quality for the IBM BB/LX system. As a measure of productivity, the reuse rate was calculated by comparing the number of lines of reused code to the project’s or subproject’s total code. The result was that for subprojects, the reuse rate was about 50 percent, and for projects, it was between 10 and 25 percent. As a measure of quality, the number of errors per line of code was calculated. The quality of the code of reused components was about nine times better during function test and 4.5 times better during system test than the code developed from scratch.

Prieto-Diaz and Freeman [9] report evaluations of three aspects of the Softech system: retrieval, classification, and ranking. The conclusions was that for retrieval effectiveness their particular scheme can reduce recall by more than 50 percent and improve precision of retrieval by more than 100 percent. Also, expansion of a query to additional queries which are “conceptually close” can have the same effect. The classification scheme was checked by 13 graduate students for ease of use, accuracy, and consistency for a set of five programs. Most students classified components correctly. To evaluate the ranking system, they asked six participants to rank five candidate programs. Their ranking was in the same order as the system ranking, but the authors could not draw conclusions because of the relatively small size of the collection and the limited number of participants.

Pict [13] is an interactive graphical programming environment using a flowchart metaphor. Pict was tested on 60 undergraduate and graduate student users at the University of Washington. The authors asked 16 questions of the users to measure usability and satisfaction. They used the experiments to get feedback about good and bad aspects of the system in order to improve it. However, they could not analyze their data with statistical tools, since they did not divide their subjects into groups so that they could compare the relative success of Pict as opposed to a conventional programming language like Pascal.

Several experiments on program readability have been reported. Tenny [20] compared the effects of procedure format to those of comments on the readability of a PL/I program. A survey with 148 experimental observations revealed that comments were the most influential factor in measuring readability, even though the program used was apparently not large enough to validate this effect. Program structures, data structures, documentation, etc.

greatly affect the readability of the programs for maintenance purposes [11], [19], [10], [17].

II. Conceptual Framework

The conceptual framework for ROPE consists of two parts:

1) the concept basis of the CODE programming environment, and

2) the concepts utilized by ROPE for classifying, finding, understanding, modifying, and combining components.

A. The CODE Programming Environment

Space precludes a full definition of CODE. Only the points of particular importance to the comprehension of ROPE are given herein. Details can be found in the users’ reference manual [11] or in [4]. CODE expresses a computation as a generalized dependency graph where the nodes are units of computation and the edges are dependency relations among the nodes. The programming language for CODE is an integrated graphical/textual representation of generalized dependency graphs. Programs are prepared by drawing generalized dependency graphs on the screen of a workstation with using elements chosen from a palette. Definitions of the nodes and arcs are completed by filling out forms. The software components (the code or design) modules which execute at a node) may be obtained either by direct coding through a text editor, as files from the UNIX file system, or through the use of ROPE. CODE components may have three kinds of dependencies: input data dependencies, output data dependencies, and exclusion dependencies. Components can interact only through the dependencies; the computations themselves are hidden. Provision is made for separate specification of the conditions (the firing rule) leading to the execution of a component and for distribution of the data generated by a component to its successors. The CODE interface provides a uniform definition of units of computation across levels of abstraction; any node may be a code module or an arbitrary subgraph. CODE accepts code modules written in a multiplicity of sequential programming languages. The formal graphical model of CODE promotes reuse in several ways. The specifications for components are rigorously separated from specification of the structuring relations among components. As an aid to construction of programs, a formal calculus of composition of components can be constructed [5] for the declarative programming language of CODE. More generally, it appears that parallel programs may be more influenced by reuse than sequential programs, since replicated use of standard modules plays an important role in the attainment of high degrees of parallelism.

B. The Conceptual Basis for ROPE

The CODE environment offers an attractive platform on which to construct a reusability system. However, it is still necessary to address the usual reuse issues of classi-
fying, finding, understanding, modifying, and combining components. Our ultimate goal is to have the role of a software designer to be to determine an appropriate set of software components, organize them with correct logic, and connect them through appropriate dependencies. In COE/E, a design may embrace components at several levels of resolution. Therefore ROPE had to meet the challenge of reuse of components spanning levels of abstraction, the challenge of understanding, modifying, and combining components which may be arbitrary subgraphs and the challenge of management of code modules in several languages including Ada, Fortran, C, and Pascal.

The sections which follow define, describe, and illustrate the use of the component classification methods which are at the core of ROPE including a new classification method called structured relational classification and the capabilities provided for understanding of components, particularly design level components. The methods for modifying and combining of components are not evaluated in the experiments reported here, and so are not discussed. (A complete discussion is found in [14].)

1) Component Classification and Finding: A fundamental problem of reusability is to find useful components in a collection of components. Convenient searching methods are necessary in order to reuse existing components. A key factor in finding the components we want is classifying them correctly. An effective classification structure is essential to the design of an effective retrieval system. Traditional classification schemes are hierarchical [12], [7] and keyword-based [61], [9]. The weak point of hierarchical classification is the difficulty of representing relationships that are more complex than many-one relationships between two entity sets. The weak point of a keyword-based scheme is that when the number of components grows, too many components are grouped in the same class. Search methods associated with the traditional classifications are appropriate to their organization. For example, in a hierarchical organization, the user can trace a single attribute tree or lattice structure in depth first fashion. It is difficult, however, to search across the tree. Also, in the traditional classification method, changing the organization of the data is a complex task.

We have designed and implemented a new software classification method, called the structured relational classification method [15]. This method potentially offers the browsing capabilities of a hierarchical organization and the flexibility and ease of reorganization of a relational method. There are two major parts to this classification method: relational and structured, as shown in Fig. 1. The relational part (the parallelogram in Fig. 1) is provided by a flat table of components and their attributes. The structured part (the various tree and lattice structures in Fig. 1) comes from assuming that the value domain of an attribute has some additional structure which flows from the semantics of the domain. These structures may be lattices, linked lists, networks, etc. This leads to two different types of activity during searching: navigating through the structured values of a specific attribute or selecting all components whose attribute values (partially) match a given set of values.

Three important goals are accomplished by this classification method: a variety of search methods, convenience of use, and easy maintenance. Variety of search is important because users have different styles of searching and may begin their search with different kinds of initial information.

Convenience matters because even libraries of useful objects remain untouched if users perceive the search techniques as unwieldy. The examples of the use of ROPE given in the next section illustrate the use of the structured relational classification methods. Finally, from a system perspective, it is often necessary to perform maintenance on the data by inserting, modifying, or removing components and attributes.

2) Understanding Components: The most common method for understanding of components, providing user written documentation for each component in the library, is accomplished here by saving the user written documentation into the database along with the CODE component graph. Design analysis of existing software components can also help the user to understand the component by abstracting key information characterizing the design component. The information which is stored for design components includes tables of: a) interface dependencies, b) data dependencies, c) exclusion dependencies, d) parallelism, and e) program hierarchy.

The interface dependencies of a component act as pins of a hardware component to which wires are attached for the movement of data into and out of the component. When using a commercially available chip as a component in a printed circuit board, a hardware engineer need only understand the function of the chip and the properties
of each input–output pin. ROPE provides this level of abstraction by displaying key information about the input/output dependencies in both graphical and tabular form, so that the designer can easily identify and connect the interface dependencies of the component to its surrounding objects in the CODE graph.

The interface dependency table contains the information needed by CODE to couple a component into a generalized dependency graph program structure: 1) interface dependency names, b) instances, the number of instances in case of a multiple dependency, c) data structure, which may be of scalar type (integer, real, boolean), array of scalar, or generic, d) interfaced object, the name of the node in the component graph connected to the interface dependency, e) I/O parameter number, the input–output parameter position of the interface dependency in the component.

The data dependency relation table shows the relationship between the nodes of a subgraph component. It also analyzes the component CODE graph to extract loops using a depth-first-search algorithm. Names of the nodes connecting to interfaced dependencies are also shown. The purpose of performing this type of analysis is to extract data dependency relationships between nodes in order to help the user better understand a software component. This makes it possible for the user to modify the component. For each node in the graph, the data dependency relation table shows: a) node name, b) instances, in case of a multiple node, c) a list of the destination nodes of its outgoing dependencies, d) a list of the source nodes of its incoming dependencies. In addition, there is a list of all input nodes (nodes receiving data from input data dependencies) and a list of all output nodes (nodes sending data through output interface dependencies).

The exclusion dependency table displays information about exclusion dependencies found in the component. For each exclusion dependency, the table shows: a) exclusion dependency name, b) data structure, c) exclusion constraints, and d) nodes participating in the exclusion dependency. Other tables describe the parallelism available within the subgraph and the hierarchical structure of the program.

III. AN EXAMPLE OF THE USE OF CODE AND ROPE

The experiments can be understood only if the reader has some feel for the properties of the programming environment. For that reason we give a substantial example of the use of CODE and ROPE. The example carries program development via component reuse through two levels. The first step is to build a program for realization of the convex hull of a set of points in the Cartesian plane from existing components and then to use the resulting program as a component of a program to divide a set of points in the Cartesian plane into four disjoint sets such that the number of points in each set is as near to equal as is possible.

A. Convex Hull

The convex hull of a set of points $S$ in a Cartesian plane is a set of arcs, formed by joining appropriate members of $S$, which defines a convex polygon enclosing the given set of points. There are several parallel algorithms for finding the convex hull. The algorithm we choose finds the points of minimum and maximum $x$ and $y$ and uses these four points as the starting points for construction of convex hull. Appendix A gives this parallel algorithm in detail.

In the following paragraphs we illustrate the use of ROPE by solving this problem. First we open the query screen and use a query to find some potentially useful components. We should not use too many attributes in the first query; otherwise, the query may be too specific and the system may not retrieve enough components. After seeing the retrieved components, we can decide on our next action. If a large number of components have been retrieved, additional qualifications are given in a next query to decrease the number of components. If a small number of components have been retrieved, we can then look at the properties of the components to find the most appropriate ones. For example, we might issue the following query {chosen for its ability to show query features}:

```
count where function = hull, data_structure = array, 
  field = mathematics, size < 100 locs, 
  keyword = line $ distance $ point $ hull
```

From this query, we get several components. If the number of components is large, we can issue an additional query:

```
set scope where function = hull, data_structure = array, 
  field = mathematics, size < 100 locs, 
  keyword = line $ distance $ point $ hull
```

After applying this query, there are now only a few candidates. Among them is the component named "half_moon_hull". Fig. 2 shows the display for this "software chip" in the CODE/ROPE environment.

We can now examine the "half-moon-hull" component. There is a variety of information about this component which may be needed, depending upon the situation. For example, sometimes we need to understand only the interface, and other times we need to trace the flow within the component. Both of these are available in ROPE. The specification of the program is also available (see Appendix B). After reading the specification, we know this program is exactly what we want. So, this component may be attached to a new convex hull program as in Fig. 3. In this program, the node "READ" reads all points (data) on the $x$-$y$-plane. "CORNERS" determines four corners and four lines. Four processors, all running HALF_Moon_Hull.’s, find a subset of the points
which, if linked by arcs, determines a convex hull for all of the points in one quadrant. "ANSWER" collects all of the determined points as a result. "WRITE" prints out the result.

After building this "convex-hull" program, it is saved into the software base. At that time several types of information related to the program are also stored. Some information is created by ROPE from analysis of the graphical/textual information defining the program. Other information, such as keywords and narratives, must be supplied by the analyst.

B. Construction of a Point Set Partition Algorithm

The next step carries the process one step further by using the convex hull program as a subgraph in the construction of a program for another algorithm which is defined as follows.

There is a set of points in the $x$-$y$-plane. Divide these points up into 4 regions. The difference of the number of points between any two regions should be less than or equal to one.

Fig 4 contains a design for a program solving this problem: it requires finding a "sort" function and a "convex hull" function to reuse in this program. We find suitable "sort" and "convex hull" functions, and then we attach them to the new program.

IV. EVALUATION OF ROPE

The initial goal in evaluating ROPE was to determine whether or not the system actually helps increase productivity and quality in the program development process.
Fig. 3. Parallel convex hull program.

Fig. 4. Parallel divide_region program.
The results from these initial experiments have guided us to designs for a continuing series of experiments. In particular, the experiments whose results will be reported herein investigated:

1) the rate of reusability achieved in the CODE/ROPE environment.
2) effectiveness of design decisions made in the construction of the system, especially in the areas of retrieving and understanding components.
3) the effect of reusability on development time and error rates.

The sections which follow give the design of the experiments and report summaries of the statistical data derived from the experiments.

A. Experimental Design

1) Population: The 43 experimental subjects were graduate students and undergraduate seniors in computer science and computer engineering. All of the subjects had considerable experience in preparation and debugging of programs in computer science or computer engineering courses. None had extensive experience with reuse-oriented programming environments or with development of parallel programs. A total of 68 programs were developed: 25 using CODE only and 43 using both CODE and ROPE.

2) Algorithm Complexity and Program Size: The programs are all executable representations of parallel algorithms taken from textbooks or the recent literature. The algorithms (Appendix C) range from the readers-writers problem to complex maze routing algorithms intended for VLSI layout application. The algorithms are modest in size although sometimes very complex in structure. The program size ranges from a few hundred lines to several thousand lines.

3) Experiments: After some preliminary training in the use of CODE, each subject developed several programs; in 25 cases they used CODE alone and for the other 43 cases they used both CODE and ROPE. The components and programs developed through the use of CODE alone were saved in ROPE for possible future use. The software base for the experiments also included programs written by the ROPE and CODE developers and earlier graduate classes which had used CODE. There were small incremental changes in the size of this library in the course of the experiments; it contained approximately 70 software components. Problems were assigned in such a way that subjects could not reuse components from the algorithms which they had developed themselves. It was the case that if the subject using CODE and ROPE together choose an appropriate design for the program, then the necessary components could be found in the component database. Individuals using only CODE had no effective access to the library of reusable components.

The distribution of problems is given in Table I.

4) Data Recording: ROPE was designed and implemented with experimental evaluation as a part of its intended functionality. ROPE incorporates automatic analysis tools to capture design and reuse features of the programs generated under its control. Additionally, for each program, subjects filled out questionnaires about the development process and their perception of various design features. These metrics are discussed in detail below.

5) Data Analysis: Descriptive statistics were generated to describe rates of reuse and user perceptions of usability. In addition, inferential statistics were used to compare the effectiveness (with respect to development time and error rates) of the CODE/ROPE combination as opposed to the use of CODE alone. These statistics show that ROPE significantly reduces both development time and error rates in the CODE environment.

B. The Metrics and Results

1) Reuse Rates: The reuse rate indicates the degree to which existing code and designs are used in a new program. In defining the reuse rate, we differentiate between the reuse of design and the reuse of code. When a program is generated from a UCG there are really two sources for the generated code. Some code is generated by CODE to implement the graph structure; this is the graph code and the number of lines generated is denoted by LOC_g. LOC_f is called the design size. Other code is inserted directly from the nodes; this is the node code and the number of lines generated is denoted by LOC_n.

Automatic counting tools embedded in ROPE were developed to calculate the design size and the code size of a program. If a user modifies the reused part of the design, the code generated from the modified parts is counted as new. If the user modifies a reused line, it is counted as a new line. Code size is defined to be the number of non-blank physical lines of the program including comments. Multiple or partial statements taking one physical line are counted as one line. Finally, if C is the total number of lines of code in a system, we write R(C) for the number of reused lines in the system.

With these conventions we make the following definitions: the total lines of code generated LOC is

\[ \text{LOC} = \text{LOC}_g + \text{LOC}_n \]
the reuse rate of design $r_D$ is
\[ r_D = \frac{R(LOC)}{LOC} \]
and the reuse rate of code $r_C$ is
\[ r_C = \frac{R(LOC)}{LOC}. \]

The reuse rate of a program is a weighted combination of the reuse rate of design and the reuse rate of code. The reuse rate of a program is
\[ r_P = \delta r_D + \epsilon r_C, \]
where $\delta$ and $\epsilon$ are weight factors of the development phase. The weighting of the design effort and the code effort in program development is different among models. For example, in the COCOMO [2] model, the weights are 0.57 and 0.43. In the Boeing model [2] they are 0.42 and 0.58. In this paper we adopt the COCOMO weighting; that is, $\delta = 0.57$ and $\epsilon = 0.43$.

Reuse rates were extremely high for the 43 programs done using ROPE. The means and 95 percent confidence intervals for those means are shown in Table II.

2) Effectiveness of the Retrieval and Understanding Mechanisms:

a) Precision of Retrieval: For each program developed, the user recorded:
- $NC_r = \text{total number of components retrieved in all queries}$.
- $NC_r = \text{number of relevant components among the retrieved ones}$.

These are used to define the precision of retrieval $P$ for each program as
\[ P = \frac{NC_r}{NC_i}. \]

The mean of the precision rates of retrieval for the 43 programs developed using ROPE was 68 percent with a 95 percent confidence interval of [60 percent, 75 percent].

b) General Information Sources for Program Understanding: Users ranked the usefulness of the following sources of information about the program on a scale of VL (Very Low—1) to VH (Very High—5): General Attributes, Source Program, Documentation, Reuse Structure, Design Analysis, Hierarchical Structure. This data is given in Table III.

3) Effect of Reusability on Productivity and Quality:

a) Development Time: To measure the effect of reusability on productivity, we use total development time. The basic data for development time comes from the users. They specified:
- $T_W$ = total development time for a program without the reusability system.
- $T_I$ = total development time for a program with the reusability system.

The data collected is summarized in Table IV.

A one-way ANOVA was performed for each of the Convex Hull, Readers/ Writers, and Producer/Consumer data sets. This test revealed that differences in mean development time between the CODE and CODE/ROPE data is significant at the alpha = 0.01 level for all three programs. In short, both inspection and statistical analysis of the data reveals that ROPE has a significant effect on development time.

b) Error Rate: Error rates are used to measure quality. Errors are counted whether they are compile errors, execution errors, or logic errors. Users supply the following information:
- $NERR_{re}$ = the number of errors in the reused part.
- $NERR_{nd}$ = the number of errors in the newly developed part.

The data collected is analyzed in Table V.

A one-way ANOVA was performed for each of the Convex Hull, Readers/ Writers, and Producer/Consumer data sets. This test revealed that differences in mean numbers of errors between the CODE and CODE/ROPE data is significant at the alpha = 0.1 level for the Convex Hull and Producer/Consumer programs. The differences were significant at the alpha = 0.05 level for Readers/ Writers. This reflects the fact that one individual made a large number of errors (35) in the solution of Readers/ Writers in the CODE only case; thereby increasing the variance of that data. The statistical evidence suggests that the use
of ROPE reduces error rates, but the picture is much less clear than it is in the development time data. This is partly caused by difficulties in collecting data on errors and also by the distinction between design and code errors. For this study these two types of errors have been treated identically. There should perhaps be some heavier weighting of design errors; certainly one would normally do this since bad designs may lead to large amounts of bad code. However, one might argue, as we did, that design in a system like CODE/ROPE is in many ways identical to programming and that errors which may affect large parts of the system may in fact be fixed relatively easily since we are programming in the design language.

V. Conclusions and Further Experimentation

This initial set of experiments has clearly established the effectiveness of CODE and ROPE in promoting component reuse in programs of modest size and complexity and in delivery of nearly error free programs with relatively little effort. There was a high correlation between all of the measures of effectiveness such as reuse rate, development time, and decreases in number of errors. These results, while satisfying and necessary as a basis for further experimentation, are by no means the conclusion of this research program on reusability.

There are subtle questions not answered by these experiments such as the relative effectiveness of different classification methods and the relative importance of factors such as the model of parallel computation and the classification method in attainment of success in reuse. CODE and ROPE clearly reinforce each other when used together, and while these experiments certainly establish the effectiveness of reuse in this situation, they do not clearly delineate the individual contributions of each system. There is also a need to investigate the impact of program scale by building and rebuilding programs of tens of thousands if not hundreds of thousands of lines. Experiments on this latter scale are beyond the resources and the competence of university researchers. We explicitly solicit the collaboration of industrial and government research laboratories to pursue these experiments.

APPENDIX A

The Parallel Algorithm of the Convex_Hull Program

Let \( S \) be the input set of points and let \( \text{CH}(S) \) be the subset of \( S \) which determines the convex hull. The goal is to find \( \text{CH}(S) \). The basic strategy is based on the following facts:

1) A line joining two points of a convex region lies completely in the region. Consequently there is a well defined idea of inside and outside for the region and its boundary.

2) The points with max \( x \) or min \( x \) or max \( y \) or min \( y \) must be included in \( \text{CH}(S) \).

3) Let \( A \) be the convex region determined by a subset, \( C \), of \( \text{CH}(S) \). Then any point \( p \) (in \( S \)) of greatest distance to the line \( L(a, b) \), among the points on the outside of \( L(a, b) \), must be in \( \text{CH}(S) \).

4) Let \( P_1 = (x_1, y_1), P_2 = (x_2, y_2) \) and \( L : y = y_0 + t*(x - x_0) \). If \( y_2 - (y_0 + t*(x_2 - x_0)) > y_1 - (y_0 + t*(x_1 - x_0)) \), then distance \( (P_2, L) > distance(P_1, L) \). Informally, the parallel algorithm is defined as follows:

1) Locate points:
   \( P_0 \) — with min \( x \)
   \( P_1 \) — with max \( y \)
   \( P_2 \) — with max \( x \)
   \( P_3 \) — with min \( y \).

2) For \( i = 0 \) to 3 let processor \( i \) execute these steps:
   a) \( j_1 := P_i \), \( j_2 := (P_i + 1) \mod 4 \), push \( P_i \) on the stack \( S_{\text{top}} \), \( \text{top}(S_{\text{top}}) := 1 \), \( \text{CH}(S) := \{ \} \)
   b) find all points in \( S \) outside \( L(j_1, j_2) \); let \( m \) be the number of such points
   c) if \( m \neq 0 \) then push on the stack a point \( P_{gd} \), among the \( m \) points which has the greatest distance from \( L(j_1, j_2) \), let \( j_1 := P_{gd} \), and go to (b)
   else add \( j_2 \) to \( \text{CH}(S) \),
   if empty \( S_{\text{top}} \), then exit this loop
   else \( j_2 := j_1 \), \( j_1 := \text{stacktop}(S_{\text{top}}) \), and go to (b)

3) Merge the elements of \( \text{CH}_i(S) \) for \( i = 0, \ldots, 3 \)

APPENDIX B

Specification of Halfmoon_Hull_Program

Component name: halfmoon_hull

Description: Given a line segment and a set of points which all lie to one side of the line containing the line segment, this program finds a subset of the points, which, if linked by arcs, determines an area containing all of the input points. This subset is called the halfmoon hull.

Input dependencies: one-dimensional array for points

Output dependencies: one-dimensional array for halfmoon hull set of points

Explanation: odd number index in array is x-coordinate even number index in array is y-coordinate
-1 in array indicates the end of data line segment is described by its two end points.
APPENDIX C

Programs Used for Experiments

1) Reader writer program—This is a traditional example program for shared data objects in the operating system area. Several readers can access the shared data object simultaneously, but writers have exclusive access.

2) Convex hull program—This problem was explained in Section III. It finds the subset of points, which, if connected by arcs, determines an area containing all of the input points.

3) Parallel_prefix program—This program computes all \( n \) initial products \( a_1 \circ \cdots \circ a_i \), \( i = 1, \ldots, n \), of a set of \( n \) elements, where \( \circ \) is an associative operation.

4) Sort merge program—This program sorts several sets of data and then merges them.

5) Producer_consumer program—This is a traditional example program for shared data objects in the operating system area. A producer process produces information that is consumed by a consumer process.

6) Shortest_path program—This program finds the path in a weighted graph connecting two given vertices \( x \) and \( y \) with the property that the sum of the weights of all the edges is minimized over all such paths.

Divide region program—This problem was explained in Section III.

REFERENCES


James C. Browne was born in Conway, AR, on January 16, 1935. He received the B.A. degree from Hendrix College, Conway, AR, in 1956, and the Ph.D. degree in physical chemistry from the University of Texas at Austin in 1960.

He worked from 1960 to 1964 as an Assistant Professor of Physics, University of Texas at Austin, carrying out quantum mechanical calculations of small molecules, and doing research and development work on operating systems. He was from 1964 to 1965, at the Queen’s University of Belfast, Belfast, Northern Ireland, on a National Science Foundation Postdoctoral Fellowship. He was, from 1965 to 1968, Professor of Computer Science and Director of the Computation Laboratory at the Queen’s University of Belfast. In 1968 he rejoined the University of Texas at Austin as Professor of Computer Science and Physics. He has served three terms as Chairman of the Department of Computer Sciences, 1968-1969, 1971-1975, and 1985-1987. He was appointed to the Regents Chair in Computer Sciences #2 in 1988. His previous research interests include operating systems, systems modeling and design, performance evaluation of computer systems, and parallel computation structures. Parallel programming is his most active and current research interest. He is Director of the University Research Initiative in Formulation and Programming of Parallel Computations. He has published about 140 refereed papers on technical subjects. He serves on numerous government committees and advisory panels.

Dr. Browne is a Fellow of the American Physical Society and the British Computer Society, and a member of the Association for Computing Machinery.

Taetje Lee received the B.S. degree in applied mathematics from Seoul National University, Seoul, Korea, in 1974, and the M.S. and Ph.D. degrees in computer sciences from the University of Texas at Austin in 1987 and 1989, respectively. His dissertation was on software reuse in parallel programming environments.

Previously he was on the research staff of the Korea Institute of Science and Technology. He is currently a member of the research staff at IBM Palo Alto Scientific Center. His research interests include object-oriented programming languages, intelligent programming environments, and reusability systems.

Dr. Lee is a member of the Association for Computing Machinery and the IEEE Computer Society.

John Werth (M’85) received the B.S. and M.S. degrees in mathematics from Emory University, Atlanta, GA, in 1962 and 1963, respectively, and the Ph.D. degree in mathematics from the University of Washington in 1968.

He is a Senior Lecturer and Research Scientist at the University of Texas, Austin. He is currently involved in research and development projects in software engineering environments with an emphasis on parallel programming and on the programming language Ada. His secondary areas of expertise are compiler technology and natural language understanding. He was an Assistant Professor and later Assistant Department Head of Mathematics at New Mexico State University from 1969 to 1975, then Associate Professor and later Chairman of Computer Science and Electrical Engineering at the University of Nevada, Las Vegas, from 1975 to 1985.

Dr. Werth is a member of the Association for Computing Machinery and the American Association for Artificial Intelligence.