

UNIVERSITY OF CALIFORNIA, IRVINE

Graphical Interface for Paradigm Oriented Distributed Computing

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in Information and Computer Science

by

Adam Chi-Lun Chang

Thesis Committee:

Professor Lubomir F. Bic, Chair

Professor Michael Dillencourt

Professor Isaac D. Scherson

2000

© 2000 Adam Chi-Lun Chang

The thesis of Adam Chi-Lun Chang is approved:

Committee Chair

University of California, Irvine
2000

DEDICATION

To

my parents and friends

in recognition of their worth

TABLE OF CONTENTS

	Page
<i>LIST OF FIGURES</i> _____	6
<i>ACKNOWLEDGEMENT</i> _____	7
<i>ABSTRACT OF THE THESIS</i> _____	8
<i>INTRODUCTION</i> _____	10
<i>THE GRAPHIC INTERFACE FOR PARADIGM ORIENTED DISTRIBUTED COMPUTING</i> _____	12
Chapter 1. Welcome Window _____	12
Chapter 2. Bag of Tasks Paradigm _____	14
Chapter 3. Branch and Bound Paradigm _____	22
Chapter 4. Genetic Algorithm Paradigm _____	30
Chapter 5. Finite Difference Paradigm _____	42
Chapter 6. Development Tool _____	53
Chapter 7. Related Research _____	55
Chapter 8. Summary and Conclusion _____	56
Chapter 9. Future work and further development _____	58
<i>BIBLIOGRAPHY</i> _____	59

LIST OF FIGURES

	Page
Figure 1. Submission Window for Welcome Session	12
Figure 2. Specification for Bag-of-Tasks Paradigm.....	14
Figure 3. Data-Flow Chart for Generating Bag-of-Tasks Paradigm.....	15
Figure 4. Submission Window for Bag-of-Tasks Paradigm.....	16
Figure 5. Feedback Window for Bag-of-Tasks Paradigm.....	20
Figure 6. Specification for Branch-and-Bound Paradigm	22-23
Figure 7. Submission Window for Branch-and-Bound Paradigm.....	24
Figure 8. Feedback Window for Branch-and-Bound Paradigm.....	29
Figure 9. Specification For Concurrent Genetic Algorithm Paradigm	30
Figure 10. Pseudo Code for Concurrent Genetic-Algorithm Paradigm	31
Figure 11. Data-Flow Chart for Distributed Genetic Programming	32
Figure 12. Submission window for Genetic-Algorithm paradigm.....	33
Figure 13. Feedback window for Genetic-Algorithm paradigm.....	40
Figure 14 Pseudo Code for Finite-Difference Paradigm.....	42-43
Figure 15. Submission window for Finite-Difference paradigm	45
Figure 16. Feedback window for Finite-Difference paradigm.....	51

ACKNOWLEDGEMENT

I would like to express the deepest appreciation to my committee chair, Professor Lubomir F. Bic, who has the attitude and the substance of a genius: he continually and convincingly conveyed a spirit of adventure in regard to research and scholarship, and an excitement in regard to teaching. Without his guidance and persistent help this thesis would not have been possible.

I would like to thank my committee members, Professor Michael Dillencourt and Professor Isaac D. Scherson, whose work demonstrated to me that concern for global affairs supported by an “engagement” in comparative literature and modern technology should always

ABSTRACT OF THE THESIS

The Graphic Interface For Paradigm Oriented Distributed Computing

By

Adam Chi-Lun Chang

Information and Computer Science

University of California, Irvine, 2000

Professor Lubomir F. Bic, Chair

Making computers work in parallel is a very important topic. The *Messengers* system developed by professors Bic and Dillencourt in Irvine, California offers the ability to distribute workload of a task into several small workers (CPUs) using mobile agents communication. In this way, *Messengers* can parallelize and speed up the job a program is designed to do. However, people who want to take advantage of *Messengers* system still have to learn how to program in *Messengers* environment and integrate this new concept to their previous sequential programming philosophy. One of professor Bic's Ph.D. student, Hairong Kuang, has developed an intermediate "translator", PODC (Paradigm Oriented Distributed Computing), which takes application-specific portions of the sequential program as the input and outputs a parallel version of the inputted program. The basic idea of PODC is taking the functions from sequential program, which must be written in C language, and schedule the worker computers to execute those functions according to different paradigms in order to increase efficiency. Nevertheless, the programmer who wrote the sequential program may need to adapt the original code to fit in PODC's setting. How will sequential programmers know where to

make the modifications and how they can communicate easily with PODC clearly becomes the problem at hand. This thesis provides a graphical user interface as a solution to this problem, which allows the user to specify the problem and monitor its progress during execution.

INTRODUCTION

A Graphical User Interface (GUI) is designed to make a computer and the operating system easier to use, but it may not be the most efficient method for a user to let computer follow his/her commands due to the tradeoff for being user-friendly. For example, computer knows what to do right a way through a command line type of interface because it can recognize the command fast and this type of interface wastes very few resources of a computer (like memories). But on the other side, graphical user interface does not have those privileges because it has to take many more resources of a computer (like memories) and it has to be able to recognize aliases of the commands in order to be user-friendly. However, graphical user interfaces always provide a pictorial means for a user to interact with an application. An application represents its data using various types of graphical objects and the user interacts with the application by manipulating the properties of these graphical objects (e.g., position, size, color, visibility, etc). This paper discusses a graphical user interface of PODC in the following chapters.

Most good GUIs include the following features: a pointing device often in the form of a mouse; some drop-down menus and radio buttons operated by the pointing device; text fields that used keyboard to interact with the users. A task is represented by a sequence of several windows of graphical images that inform the users what the computer is doing and allow the users can tell the computer what to do. The GUIs used in PODC basically contain all the qualities that a good GUI should have. However, there are four different paradigms that PODC can help converting from sequential to parallel

style and different philosophies underneath of each paradigm to make vary the converting flow. As a result, each interface is designed differently to allow each paradigm to meet its designed purpose yet maintain its user-friendliness.

The interface will let user choose which paradigm he/she wants to use, and provide by a sequence of windows for the user to input the required data for parallel computing of the chosen paradigm. The detail illustrations are in the chapter 2 (Bag of Tasks paradigm), chapter 3 (Branch and Bound paradigm), chapter 4 (Genetic Algorithm paradigm) and chapter 5 (Finite Difference paradigm). PODC will return the computing result to the user as soon as it comes out. However, if the result is not ready in a short time (under several hours), then user may choose to quit the PODC first and retrieve the data later. PODC will notice the user is quitting without getting the final result come out, so a task ID for the task will be given to the user when the user is quitting so that the user can use this ID to retrieve the result. Besides, the user will also get an e-mail that notifies the job is done if he/she never retrieves the result.

THE GRAPHIC INTERFACE FOR PARADIGM ORIENTED DISTRIBUTED COMPUTING

Chapter 1.

Welcome Window

This window allows the user to choose whether to start a new task of distributing a sequential program or check the results of previous tasks. The radio button for Check Status (Task ID Number and E-Mail Address) will be disabled by checking New Task, and vice versa. The user can choose which paradigm he/she wants to use by choosing the choices of drop-down menu from Paradigm Choice. Bag of Tasks, Branch and Bound, Genetic Algorithm and Finite Difference are the four paradigms currently available in PODC.

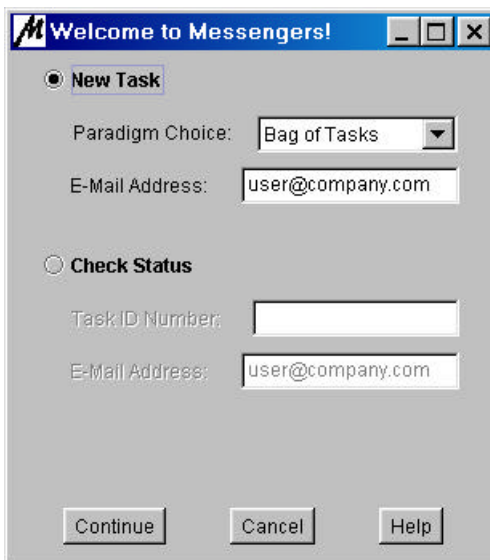


Figure 1. Submission Window for Welcome Session

If the user chooses to check status of previous tasks, he/she needs to input the task identification number obtained when he/she quits PODC. The user also needs to input the email address so that POCD can have a check to see if the task identification number

is matched with the email address in order to prevent someone retrieving result of other user's task. The reason why PODC will do this kind of checking is because once the result has been retrieved, PODC will not keep a copy of result on the server side.

Chapter 2.

Bag of Tasks Paradigm

2.1. Paradigm Specification

The bag-of-tasks paradigm applies to the situation when the same function is to be executed a large number of times for a range of different parameters. Applying the function to a set of parameters constitutes a task, and the collection of all tasks to be solved is called the “bag of tasks”, since they do not need to be solved in any particular order. At each iteration, a worker grabs one task from the bag and computes the result.

```
1. BagOfTasks()
2. {
3.     Data * D;           // user-defined type for the initial problem data
4.     struct Task *T;    // struct type describing a task
5.     struct Result *R;  // struct type for describing the output of executing a task
6.     FILE * FP;        // output file handler

7.     D = Data_Initialization_Function();
8.     for (Iteration_Count) do
9.     {
10.         T = malloc(sizeof(struct Task));
11.         T = Generate_Task_Function(D, T);
12.         add T to the bag of tasks BT;
13.     }
14.     while (BT is not empty)
15.     {
16.         T = remove a task from BT;
17.         R = malloc(sizeof(struct Result));
18.         R = Compute_Function(T , R);
19.         Output_Function(FP, R);
20.     }
21. }
```

Figure 2. Specification for Bag-of-Tasks Paradigm

Figure 2 shows the structure of the bag-of-tasks paradigm. The identifiers in bold face identify the application-specific components that must be provided by the user through submission window. The first step PODC will do is to initialize the problem data

and initial state of task generation (line 7). Then the bag of tasks (BT) is created, which is accomplished by the loop in lines 8-13. The iteration count represents either a fixed number of iterations or is given implicitly by reading input values from a file until the end of file is reached. The while loop on lines 14-20 represents the actual computation, which is repeated until the bag of tasks is empty. Multiple workers may execute the loop independently. All workers have shared access to the task bag and the output data. Each worker repeatedly removes a task (line 18), solves it by applying the main compute function to it (line 19), and writes the result into a file (line 20).

The process of generating the bag of tasks is presented in Figure 3. At first the data initialization function generates the problem data, which consists of the static problem data and the initial state of generating task. The reference to the problem data is then passed to the task generation function, which generates a new task. The process is continued until the iterative count is satisfied. Whenever the task generation function is called, the static problem data remains the same, but the state of generating tasks is updated, thus creates a new task.

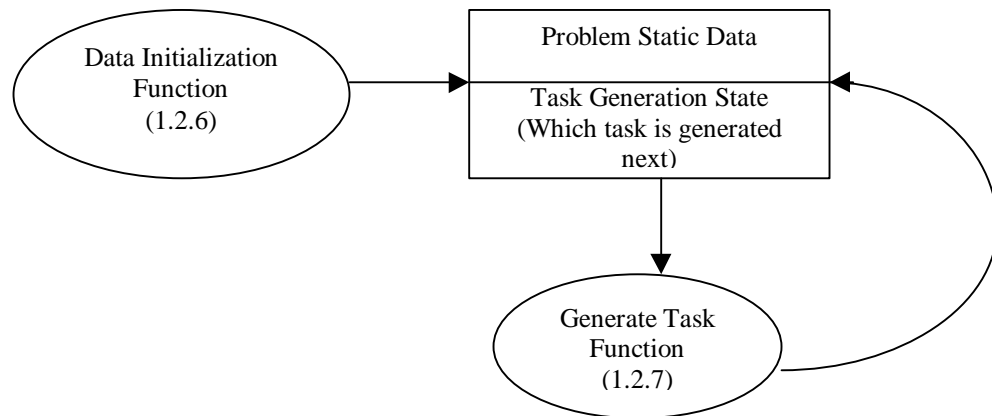
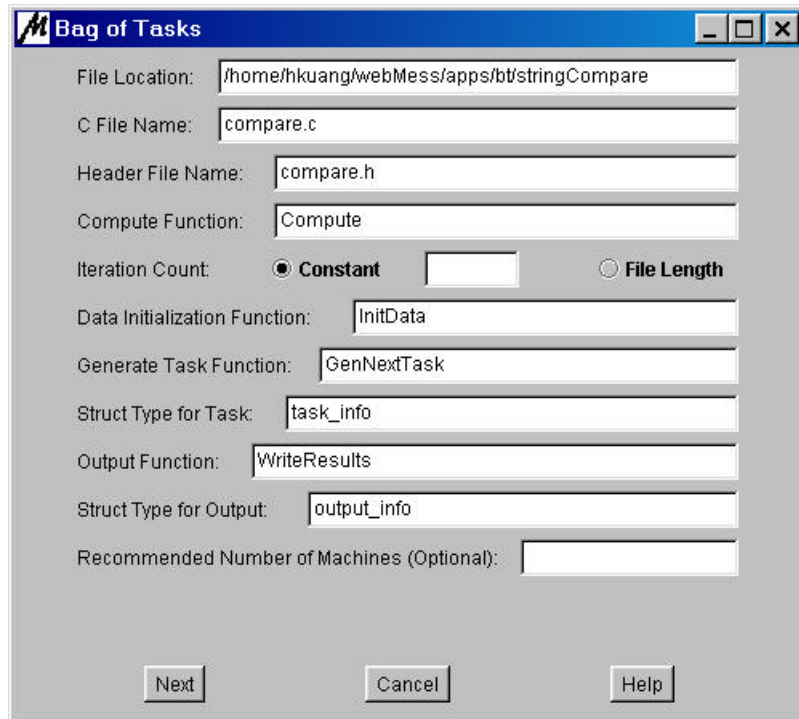


Figure 3. Data-Flow Chart for Generating Bag-of-Tasks Paradigm

2.2. Problem Specification: Submission Window



The screenshot shows a window titled "Bag of Tasks" with the following fields and controls:

- File Location: /home/hkuang/webMess/apps/bt/stringCompare
- C File Name: compare.c
- Header File Name: compare.h
- Compute Function: Compute
- Iteration Count: Constant File Length
- Data Initialization Function: InitData
- Generate Task Function: GenNextTask
- Struct Type for Task: task_info
- Output Function: WriteResults
- Struct Type for Output: output_info
- Recommended Number of Machines (Optional):

Buttons at the bottom: Next, Cancel, Help

Figure 4. Submission Window for Bag-of-Tasks Paradigm

Figure 4 shows the graphic interface used to specify bag-of-task problem. At the top of the window, the user provides the location of the sequential source programs, and the name of the header file(s) and the program file(s), which defines the application-specific components. Next, the user specifies the application-specific functions and data structures, as identified in bold font in Section 1. Finally, the user can specify a recommended number of machines to use in the computation. In the following subsections, each application-specific component will be explained in detail.

2.2.1. File Location [Text Field]

The user needs to input the path where they store their sequential Bag of Tasks program(s), header file(s), and the input data file(s). No subdirectory is allowed under this directory.

2.2.2. C File Name [Text Field]

The user needs to input the name of the .c file, which contains function components of the sequential Bag of Tasks program. The C program file(s) are assumed to be stored in the directory PODC gets from File Location (2.2.1), and therefore the path names are not required to be specified.

2.2.3. Header File Name [Text Field]

The user needs to input the name of the .h file (header file), where the struct type for Task (2.2.8) and Output (2.2.9) are defined. The user might have to write this file specifically for submitting programs since there are certain restrictions to this file. First, no header files are allowed to be included in this file. In addition, no macros and dynamic type variable (like pointers) declaration are allowed.

2.2.4. Compute Function [Text Field]

The user needs to input the name of Compute function, which is the main compute procedure of the sequential Bag of Tasks program (It is not the main function in standard C). This Compute function executes each task that is generated by Generate Task function (2.2.7), and returns the output, which will then be written to the output file by Output function (2.2.9).

Prototype: **struct Result* compute(struct Task* task, struct Result* result)**

Input: A pointer to a task whose type is describes in (2.2.8), and a pointer to the result that describes the output after executing a task (2.2.10). If the result pointer is *NIL*, the space for storing the result will be allocated inside the function. Otherwise, the output will be stored in the space where the result pointer points.

Output: A pointer to result whose type is defined at Struct Type for Result (2.2.10). If no output is returned, a NIL will be returned. Otherwise the pointer that points to the output is returned.

2.2.5. Iteration Count

Iteration Count defined the termination condition of the while loop which generates the task bag. There are two types of iteration number, constant and file length. These two choices are exclusive to each other, so if users choose *File Length*, the text field for *Constant* data input will be disabled automatically to avoid confusion.

Constant [Radio Button + Text Field]: If the iteration number is fixed, the user needs to click the radio button before “Constant” and inputs the constant number.

File Length [Radio Button]: If the input data is read from a file and the iteration number depends on the length of the input file, the user needs to click the radio button before “File Length”.

2.2.6. Data Initialization Function [Text Field]

The user needs to input the name of Data Initialization Function that generates all the input data, which includes the static problem data and the initial state of task generation. It returns the reference to the input data, which will then pass to the Generate Task Function (2.2.7) to use. The input data may also be an input file handler. The initialization function opens the input file and returns the file handler.

Prototype: **void* DataInit()**

Input: None.

Output: A pointer to the input data.

2.2.7. Task Generate Function [Text Field]

The user needs to input the name of Generate Task function, which produces a new task whenever it is called. The state of task generation is recorded in the input problem data that is modified every time a new task is generated. This is the function that partitions the whole application into multiple tasks.

Prototype: **struct Task* GenNextTask(void* data, struct Task* task)**

Input: A pointer to an input data, and a pointer to next task whose type is declared at Struct Type for Task (2.2.8). If the task pointer is NIL, the memory space for storing a new task is allocated inside the function. Otherwise the space will already be allocated, hence information describing the newly generated task will be stored in this space.

Output: A pointer to next task whose type is the user-defined struct type describes in Struct Type for Task (2.2.8). It returns NIL if no next task is generated, otherwise it returns the pointer to the newly generated task.

2.2.8. Struct Type for Task [Text Field]

The user needs to input the name of struct type, which describes a task. Besides the input data for executing a task, this data structure may also include the task number information, which might also be delivered to the Result data structure.

2.2.9. Output Function [Text Field]

The user needs to input the name of output function, which saves the result generated by executing a task in a file. The assumption for this function is that the output file is open and ready for write.

Prototype: **int WriteResult(FILE*, struct Result*)**

Input: An output-file handler, which records all the results, and includes a pointer to an output data, whose type is declared at (2.2.10).

Output: The error number. 0 means normal execution, while -1 means that errors occur while the data is writing.

2.2.10. Struct Type for Result [Text Field]

The user needs to input the name of struct type that describes the output data generated by solving a task. The information of the task, which generates this result, might also be described in this data structure. The result is then written into the output file. The user might use the task information to integrate the output data.

2.2.11. Recommended Number of Machines [Text Field]

This field is optional for the user. If a user knows the optimal number of machines to run the program, he/she can input the number. Otherwise the user can just leave it blank to let the system decides. (Usually we choose as many available machines as possible.)

2.3. The Feedback Window



Figure 5. Feedback Window for Bag-of-Tasks Paradigm

While the application is running, the user and the running application will be able to interact. Figure 5 shows the feedback window of bag-of-tasks application, which communicates the running status of the application with the user.

At the very top of the window, a moving icon indicates that the application is currently running. Below is the text that describes error or success messages. The feedback window also displays the number of tasks being executed up to date. Since the user usually knows the number of tasks in the application, s/he can estimate the fraction of the total work that has been completed. The window also shows the number of machines currently being used by the distributed application.

Chapter 3.

Branch and Bound Paradigm

3.1. Paradigm Specification

Branch-and-Bound search is applicable to various combinatorial optimization problems, and is generally applied when the goal is to find the exact optimum. A branch-and-bound search is described as a search through a tree, where the root node corresponds to the original problem to be solved, and each descendant node corresponds to a sub-problem of the original problem. Each leaf corresponds to a feasible solution. The search tree is constructed dynamically during the search and consists initially of only the root node. To speed up the search, a sub-tree is pruned if it can be determined that it will not yield a solution that is better than the best currently known solution.

```
1. BranchAndBound()
2. {
3.     File * F;        // file pointer to output file
4.     Integer B;      // current bound
5.     SB;             // next bound

6.     struct ProblemData D;    // struct type for problem data (internal use only)
7.     struct TaskStructName    // struct type for a node in the tree (3.2.10)
8.     R;                       // root node
9.     S;                       // solution node
10.    SN;                      // next solution node
11.    Pool L;                  // a collection set contains unexplored tree nodes

12.    B = +Infinitive Number;  // initialize the bound to an infinitive positive
                               // number
13.    D = GenProblemData();    // generate the initial problem data
14.    R = GenRootNode ( D );   // generate root node according to initial problem
                               // data

15.    if ( an improved initial bound is to be used )
16.    {
17.        S = GenInitSol ( D );
18.        B = GenBound ( S, D );
```

```

19.   }
20.   L = { R };
21.   while ( L is not empty )
22.   {
23.       N = SelectNode ( L );
24.       SN = NextBranch ( N, D );
25.       if ( SN == nil )
26.       L = L - { N };
27.       else
28.       {
29.           SB = GenBound ( SN, D );
30.           if ( !IsSol ( SN ) )
31.           {
32.               if ( SB < B )
33.               L = L + {SN};
34.           }
35.       else
36.       {
37.           if ( SB < B )
38.           {
39.               B = SB;
40.               S = SN;
41.           }
42.       }
43.   }
44.   WriteSol ( S, F )
45. }

```

Figure 6. Specification for Branch-and-Bound Paradigm

Figure 6 presents the basic structure of the branch-and-bound paradigm.

Without loss of generality, we assume the goal of the program is to find the minimum value. We assume D is the initial problem data, which is passed as a parameter to certain functions. R is the initial root node. The algorithm maintains a pool L of tree nodes that has yet to be explored. S is the current best solution, and B is the bounding value of S.

3.2. Problem Specification: Submission Window

The screenshot shows a window titled "Branch & Bound" with the following fields and controls:

- File Location: /home/hkuang/webMess/apps/backTrack/tsp1
- C File Name: tsp.c
- Header File Name: tsp.h
- Optimization Direction: Minimum Maximum
- Branch Function: Branch
- Evaluation Function: GenBound
- Initial Solution Function (Optional): GenInitSol
- Check Solution Function: IsSol
- Data Initialization Function: GenInitData
- Task Structure Name: Task
- Initial Task Generating Function: GenInitNode
- Output Function: WriteSol
- Recommended Number of Machines (Optional):

Buttons at the bottom: Next, Cancel, Help.

Figure 7. Submission Window for Branch-and-Bound Paradigm

Figure 7 shows the graphic interface used to specify a branch and bound problem. At the top of the window, the user provides the location of the sequential source programs, and the name of the header file(s) and the program file(s), which define the application-specific components. Next, the user specifies the application-specific functions and data structures, as identified in bold font in Section 1. Finally, the user can specify a recommended number of machines to use in the computation. In the following subsections, each application-specific component will be explained in detail.

3.2.1. File Location [Text Field]

The user needs to input the path (FTP format) where they store their sequential Bag of Tasks program, so that this system can locate and find those files through FTP.

User also needs to make files and directory public so that this system can use anonymous login. However, all the files of the sequential Bag of Tasks program should be placed under one single directory. Error will happen if the files are placed under more than one directory.

3.2.2. C File Name [Text Field]

The user needs to input the name of the .c file, which contains the main procedure of the sequential Branch and Bound code. The C program file(s) are assumed to be stored in the directory PODC gets from File Location (3.2.1), and therefore the path names are not required to be specified.

3.2.3. Header File Name [Text Field]

The user needs to input the name of the .h file (header file) of the sequential Branch and Bound program. There should be only one .h file (header file) existing in the program. If not, the user has to combine all the .h files (header files) into one. Error will happen if more than one .h file (header file) is used.

3.2.4. Optimization Direction

The user needs to choose the solution attribute because of the characteristic of Branch and Bound type problem. Two types of attributes (minimum and maximum) could be chosen to help define the better solution and show the correct feedback information.

Minimum [Radio Button]: If searching for minimum solution, user has to click the radio button before “Minimum”.

Maximum [Radio Button]: If searching for maximum solution, user has to click the radio button before “Maximum”.

3.2.5. Branch Function [Text Field]

The user needs to input the name of branch function, which executes the branch operation in sequential Branch and Bound program. This branch operation defines how to divide a problem into sub-problems and every time this function is called, it only returns one unique child of the parent node. Repeated calls to this function return the sub-problems and then the value NIL, with each sub-problem being returned exactly once.

Input: Parent node (in a type from Task Structure Name (3.2.10)) that wants to be branched, and the problem data.

Output: Return a child node (in a type from Task Structure Name (3.2.10)) that has not been returned yet of the parent node.

3.2.6. Evaluation Function [Text Field]

The user needs to input the name of evaluation function, which evaluates the value of a new solution in order to decide whether to prune the new solution. This is the function that computes a value for a particular node in the sub-tree. For a leaf node, which corresponds to a feasible solution, the value returned is the value of that feasible solution. For a non-leaf node X in a minimization problem, the value returned is a lower bound on the value of any feasible solution corresponding to a leaf node that is a descendent of X.

Input: A node (in a type from Task Structure Name (3.2.10)) that wants to be evaluated, and the problem data.

Output: Return a newer bound (Integer type).

3.2.7. Initial Solution Function [Text Field]

The user needs to input the name of initial solution function, which generates an initial feasible solution that is used to provide an initial pruning bound

Input: A parameter that represents problem data. (The output from Data Initialization Function (3.2.9))

Output: A solution type variable that represents the first feasible solution.

3.2.8. Check Solution Function [Text Field]

The user needs to input the name of check solution function, which determines if a new node is an actual solution (leaf node in the branch-and-bound tree) or not.

Input: A node (in a type from Task Structure Name (3.2.10)) that wants to be checked.

Output: Return TRUE if the passed-in node is a solution, otherwise return FALSE.

3.2.9. Data Initialization Function [Text Field]

The user needs to input the name of data initialization function, which initializes the user input data and returns a data variable to Initial Task Generation Function (3.2.11) and Initial solution Function (3.2.7).

Input: None.

Output: Initialized problem data.

3.2.10. Task Structure Name [Text Field]

The user needs to input the name of the user-defined struct data type, which contains all the input data.

3.2.11. Initial Task Generation Function [Text Field]

The user needs to input the name of initial task generation function, which generates the first task (root node in the search tree) for sequential Branch and Bound program.

Input: Problem data from Data Initialized Function (3.2.9).

Output: Root node in a type from Task Structure Name (3.2.10).

3.2.12. Output Function [Text Field]

The user needs to input the name of the output function, which will write the result to an output file.

Input: A pointer to the solution, which is the user-defined struct type (2.2.10) and a pointer to an output file, which records all the results.

Output: None.

3.2.13. Recommended Number of Machines [Text Field]

This field is optional for the user. If a user knows the number of machines, which can optimize the program, then he/she can input the data. Otherwise the user can just leave it blank to let the system decides the optimized number of machines for the program. (Usually the optimized number is as many as possible.)

3.3. The Feedback Window

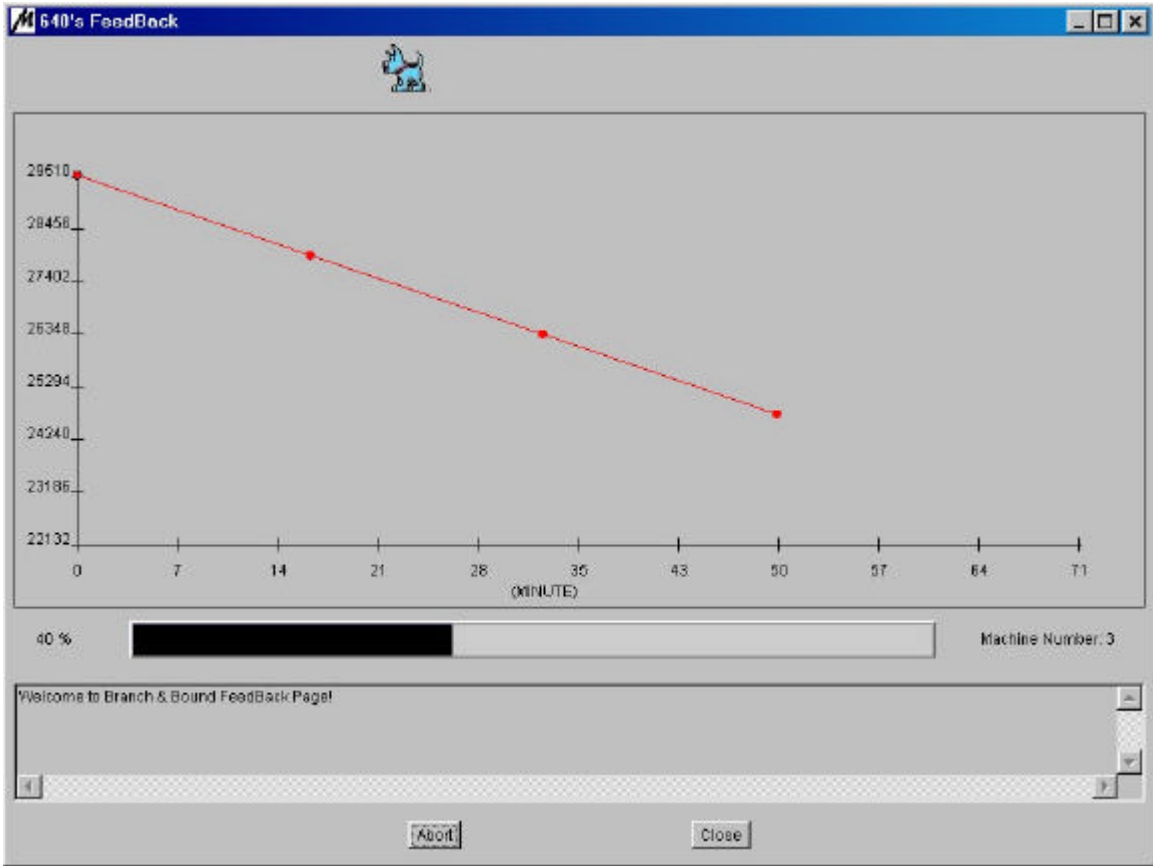


Figure 8. Feedback Window for Branch-and-Bound Paradigm

While the application is running, the user and the running application are able to interact. Figure 8 shows the feedback window of branch and bound application, which communicates with the user about the running status of the application.

At the very top of the window, a moving icon indicates that the application is still running. Below that is the text that will describe error or success messages. The window also shows the number of machines currently being used by the distributed application.

Chapter 4.

Genetic Algorithm Paradigm

4.1. Paradigm Specification

The genetic programming paradigm also solves optimization problems by using the Darwinian principles of survival and reproduction of the fittest, and genetic inheritance. Unlike Branch and Bound paradigm, the Genetic Algorithm paradigm is generally applied to find a good but not necessarily optimal solution.

Here are some names and meanings of the variables used in the following pseudo codes.

D: problem data

P: Population

N: Neighbor

M: Immigrant

I: Individual

F: Fitness score

BR: Best individual

BF: Best fitness score

```
1. SeqGenetic()
2. {
3.     Data * D;      // user-defined type for initial problem data
4.     Population * P; // user-defined type for population
5.     D = GenProblemData();
6.     P = GenInitPop( S, D );
7.     Do
8.     {
9.         P = CreateNextGen( P, D );
10.    } Until (Termination_condition);
11.    I = BestIndividual( P );
12.    WriteSol( I );
13. }
```

Figure 9. Specification For Concurrent Genetic Algorithm Paradigm

Figure 9 shows the basic structure of a sequential genetic programming paradigm.

The first step generates the static problem data (line 3) and then randomly creates an initial population P of size S (line 6). The while loop on lines 7-10 represents the

evolution process. In each iteration, a new generation is created by applying genetic operations such as crossover, mutation, and reproduction (line 9). This process continues until a termination condition holds; typically, the termination condition is either based on the number of iterations completed or the quality of the best solution obtained. Finally a result is designated (line 11) and written to a file (line 12).

```

1. ConGenetic()
2. {
3.     D = GenProblemData();
4.     for (each subpopulation pool)
5.     {
6.         P = GenInitPop(PopulationSize, D);
7.         Do
8.         {
9.             for (EmigrationInterval iterations)
10.                P = CreateNextGen(P, D);
11.            for (each neighbor N of this population pool)
12.            {
13.                for (EmigrationRate iterations)
14.                {
15.                    I = Select Emmigrant(P);
16.                    send I to N;
17.                }
18.            }
19.            for (all the immigrants M)
20.                P = Replace(P, M);
21.        } Until (Termination_condition);
22.    }
23.    BF = + ∞
24.    for (each subpopulation pool P)
25.    {
26.        I = BestIndividual(P);
27.        F = Fitness(I, D);
28.        if (F < BF)
29.        {
30.            BF = F;
31.            BR = I;
32.        }
33.    }
34.    WriteSol(BR);
35. }

```

Figure 10. Pseudo Code for Concurrent Genetic-Algorithm Paradigm

Pseudo code of distributed version of genetic programming paradigm is based on the concurrent modification of the paradigm shown in Figure 10. The population is divided into multiple subpopulations, which evolve independently but occasionally exchange individuals. This concurrent scheme allows the execution of genetic algorithms to achieve higher performance through course-grained parallelism, which also provides a good quality solution by mixing individuals from different subpopulations so that the effect is that of having one large distributed population rather than a number of small, unrelated populations.

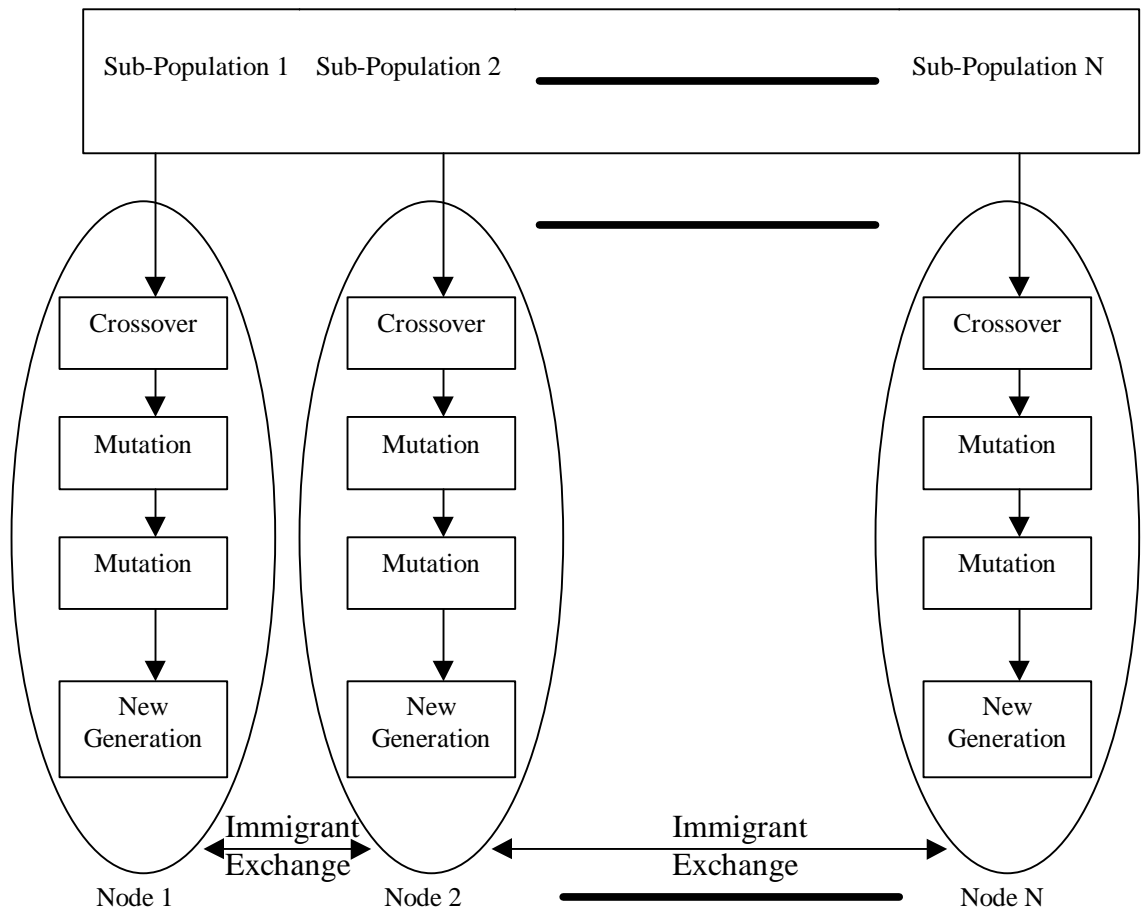


Figure 11. Data-Flow Chart for Distributed Genetic Programming

4.2. Problem Specification: Submission Window

The screenshot shows a window titled "Genetic Algorithm" with the following fields and options:

- File Location:
- C File Name:
- Header File Name:
- Struct Type for An Individual:
- Optimization Direction: Minimum Maximum
- Network Topology: Isolated Ring Grid Complete
- FUNCTIONS**
- Data Initialization:
- Population Initialization:
- Next Generation:
- Select Emigrant (Optional):
- Select for Replacement (Optional):
- Fitness:
- Best Individual:
- Output:
- Termination Methods: Constant Function
- Initial Size of Population:
- Recommended Initial Emigrant Rate (Optional):
- Recommended Initial Emigrant Interval (Optional):
- Recommended Number of Machines (Optional):

Buttons: Next, Cancel, Help

Figure 12. Submission window for Genetic-Algorithm paradigm

Figure 12 shows the graphic interface used to specify genetic algorithm problem. At the top of the window, the user provides the location of the sequential source programs, and the name of the header file(s) and the program file(s), which define the application-specific components. Next, the user specifies the application-specific functions and data structures, as identified in bold font in Section 1. Finally, the user can

specify a recommended number of machines to use in the computation. In the following subsections, each application-specific component will be explained in detail.

4.2.1. File Location [Text Field]

The user needs to input the path (FTP format) where they store their sequential Genetic Algorithm program, so that this system can locate and find those files through FTP. User also needs to make files and directory public so that this system can use anonymous login. However, all the files of the sequential Genetic Algorithm program should be placed under one single directory. Error will happen if the files are placed under more than one directory.

4.2.2. C File Name [Text Field]

The user needs to input the name of the .c file, which contains the main procedure of the sequential Genetic Algorithm code. The C program file(s) are assumed to be stored in the directory PODC gets from File Location (4.2.1), and therefore the path names are not required to be specified.

4.2.3. Header File Name [Text Field]

The user needs to input the name of the .h file (header file) of the sequential Genetic Algorithm program. There should be only one .h file (header file) existing in the program. If not, the user has to combine all the .h files (header files) into one. Error will happen if more than one .h file (header file) is used.

4.2.4. Struct Type for An Individual [Text Field]

The user needs to input the name of struct type, which describes an individual.

4.2.5. Optimization Direction

The user needs to choose the solution attribute because of the characteristic of Genetic Algorithm type problem. Two types of attributes (minimum and maximum) could be chosen to help define the better solution and show the correct feedback information.

Minimum [Radio Button]: If searching for minimum solution, user has to click the radio button before “Minimum”.

Maximum [Radio Button]: If searching for maximum solution, user has to click the radio button before “Maximum”.

4.2.6. Network Topology

Isolated [Radio Button]: It is the default network topology, which means the nodes in the Messengers System will not communicate with any other nodes. In other words, if user chooses this topology, then there will not be any exchanges of immigrants occurred at all, and the field of Recommended Initial Emigrant Rate (4.2.17) and Recommended Initial Emigrant Interval (4.2.18) will be disabled.

Ring [Radio Button]: In this topology, each node will be connected to a neighbor node from left and right direction in the Messengers System and have immigrants exchanges with them using the inputs from Recommended Initial Emigrant Rate (4.2.17) and Recommended Initial Emigrant Interval (4.2.18).

Grid [Radio Button]: In this topology, each node will be connected to a neighbor node from left, right, up, and down direction in the Messengers System and have immigrants exchanges with them using the inputs from Recommended Initial Emigrant Rate (4.2.17) and Recommended Initial Emigrant Interval (4.2.18).

Complete [Radio Button]: In this topology, each node will be connected to every other node in the Messengers System and have immigrants exchanges with them using the inputs from Recommended Initial Emigrant Rate (4.2.17) and Recommended Initial Emigrant Interval (4.2.18).

4.2.7. Data Initialization [Text Field]

The user needs to input the name of Data Initialization Function that generates all the input data, which includes the static problem data and the initial state of task generation. It returns the reference to the input data, which will then pass to the Generate Initial Population Function (4.2.8) to use.

*Prototype: structure Data * GenProblemData()*

Input: None.

Output: A pointer to problem data.

4.2.8. Population Initialization [Text Field]

The user needs to input the name for Population Initialization function that initializes the population. It takes the population size and problem data and returns a pointer to the initialized population.

*Prototype: structure Population * GenInitPop(int PopulationSize, structure Data * data)*

Input: An integer that is the size of population (the initial size is set in 4.2.15), and a pointer to problem data, which is originally defined in the output of Data Initialization (4.2.7).

Output: A pointer to initialized population.

4.2.9. Next Generation [Text Field]

The user needs to input the name of function that generates the next generation. It gets the inputs of current generation of population and problem data so that this function can generate a better next generation.

Prototype: **structure Population * CreateNextGen(P, D)**

Input: A pointer to current population and a pointer to problem data, which is originally defined in the output of Data Initialization (4.2.7).

Output: A pointer to a new generation of population.

4.2.10. Select Emigrant [Text Field]

The user needs to input the name of Select Emigrant function that select good individual from the current node then send to its neighbors for evolution process. It takes the population of the current local node and then returns a good individual. This part will be disabled automatically if user chooses to use isolated network topology because in an isolated network, no immigrants will be exchanged between nodes.

Prototype: **structure Individual * SelectEmigrant(P)**

Input: A pointer to the population in current local node.

Output: A pointer to a good individual whose type is described in (4.2.4).

4.2.11. Select for Replacement [Text Field]

The user needs to input the name of Select Replacement function that when a node receives good immigrants from its neighbor, it replaces a bad individual in original population by the best one of immigrants. This part will be disabled automatically if user chooses to use isolated network topology because no immigrants will be exchanged between nodes in isolated network.

Prototype: **P = Replace(P, M)**

Input: A pointer to the population in the current local node and the set of immigrants this node receives.

Output: A new evolved population of the current local node.

4.2.12. Fitness [Text Field]

The user needs to input the name of function that shows how good an individual fits to the problem data by a score.

Prototype: **int Fitness(I, D)**

Input: A pointer to an individual (4.2.4) and a pointer to problem data.

Output: An integer that represents the score and shows how the input individual fits the problem data.

4.2.13. Best Individual [Text Field]

The user needs to input the name of function that obtains the best individual among a population. The result this function generates will be sent to Output function (4.2.14) since it is the best solution.

Prototype: **structure * Individual BestIndividual(P)**

Input: A pointer to a population.

Output: A pointer to a best individual in the passed in population, and the type of the output is described in (4.2.4).

4.2.14. Output [Text Field]

The user needs to input the name of output function, which saves the result generated by BestIndividual function (4.2.13). The assumption for this function is that the output file is open and ready for write.

Input: A pointer to the solution, which is the Individual type defined in (4.2.4) and a pointer to an output file, which records all the results.

Output: None.

4.2.15. Termination Methods

Constant [Radio Button + Text Field]: This is the default choice. The user needs to input the number of the constant, which is the number of the iteration the program is going to evolve.

Function [Radio Button + Text Field]: If the user chooses to let the function determines when to terminate the program, then the user needs to input the name of the function.

4.2.16. Initial Size of Population [Text Field]

The user needs to input the size of initial population so that the program can keep the size of the population.

4.2.17. Recommended Initial Emigrant Rate [Text Field]

This field is optional for the user. If a user knows the best rate to exchange immigrants between neighbors in population pool, then he/she can decide the emigrant rate. Otherwise the user can just leave it blank to let the system decides the optimized number for emigrant rate (The default value is 4). However, this field will not be available if user chooses isolated on network topology (4.2.6)

4.2.18. Recommended Initial Emigrant Interval [Text Field]

This field is optional for the user. If a user knows the best interval value of exchanging immigrants between neighbors in population pool, then he/she can decide how often this parallel version will exchange immigrants. Otherwise the user can just

leave it blank to let the system decides the optimized number for emigrant interval (The default value is 100). However, this field will not be available if user chooses isolated on network topology (4.2.6)

4.2.19. Recommended Number of Machines [Text Field]

This field is optional for the user. If a user knows the number of machines, which can optimize the program, then he/she can input the data. Otherwise the user can just leave it blank to let the system decides the optimized number of machines for the program. (Usually the optimized number is as many as possible.)

4.3. The Feedback Window

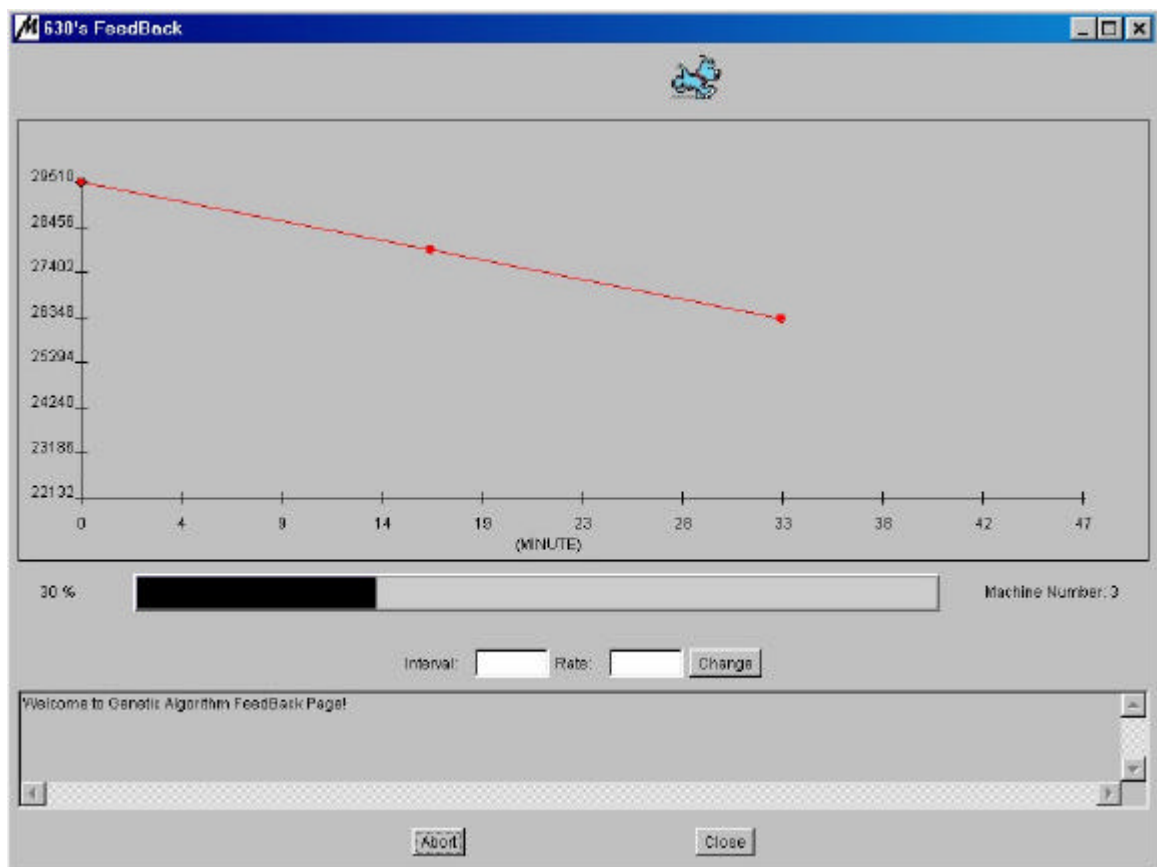


Figure 13. Feedback window for Genetic-Algorithm paradigm

While the application is running, the user and the running application are able to interact. Figure 13 shows the feedback window of genetic algorithm application, which communicates the running status of the application to the user.

At the very top of the window, a moving icon will indicate that the application is currently running. Users are allowed to change the values of interval and the rate of exchanging immigrants dynamically in this window by pressing the “Change” button after inputting in the spaces of “Interval” and “Rate”. Below that is the text describing error/success messages. The window also shows the number of machines currently being used by the distributed application.

Chapter 5.

Finite Difference Paradigm

5.1. Paradigm Specification

Finite difference method is a typical way to solve differential equations. In the finite difference method, we have a discrete d -dimensional grid of element locations, and we want to compute a value $u_{x,t}$ at each grid location x and for each time step t . The value of $u_{x,t}$ is a function of its neighbors' values at the previous time step.

```
1. FDM()
2. {
3.     double vals1[XSIZE], vals2[XSIZE];
4.     double *tempVals, *oldVals, *newVals;
5.     double ΔX = XLEN/(XSIZE-1);    // non-toroidal grid
6. // initialization
7.     for ( every element X )
8.     {
9.         vals1[X] = Init( global index of X );
10.    }
11.    oldVals = vals1;
12.    newVals = vals2;
13. // iterative computation
14.    until ( Termination_condition )
15.    {
16.        send the boundary to its neighbors;
17.        receive the boundary from its neighbors;
18.        for ( every element X )
19.        {
20.            newVals[X] = Compute( X, oldVals, Dt, ΔX );
21.        }
22.        tempVals = oldVals;
23.        oldVals = newVals;
24.        newVals = oldVals;
25.    }
26. // output
27.    for ( every element X )
28.    {
29.        WriteResult( outFile, X, oldVals );
30.    }
31. }
```

Figure 14 Pseudo Code for Finite-Difference Paradigm

Init: This is the initialization function that initializes the value of each element.

Termination_condition: This specifies the criteria for terminating the computation. It is either a fixed number or one of the tolerance predicates: L1, L2, or L infinity. The user provides the minimum tolerance.

Compute: This is the main function of the paradigm. Given its own value and its neighbors' values in the previous time step, it will produce the new value for the current time step.

WriteResult: This is an output function, which writes the value of an element to an output file.

Figure 1 shows the structure of finite difference paradigm for 1D problem. Multiple dimensional problems can be solved almost in the same way. The first difference is that arrays to store elements must be in multiple dimensions. The second difference is that each element is represented by indexes from multiple dimensions. The last difference is we need to pass in not only the length of the each interval in x dimension but also the length of each interval in other dimensions when we compute the new value for the next time step.

The algorithm starts by initiating the values of all elements (line 7-9), and pointers to element buffers (line 10-11). It then repeatedly computes the new values of all elements for each time step until the termination condition is satisfied (line 14-25). The termination condition (line 14) can be fixed number of iterations or each of three tolerance predicates, i.e. L1, L2, or L infinity. In each iteration, boundary information is sent to the neighbors (line 16) and boundary information from its neighbor grids is received (line 17). Then the value of each element gets updated (line 18-21). Pointers to

two element buffers are swapped (line 22-24). Finally each value of elements is written into the output file (line 27-30). Specifying a finite difference paradigm problem requires 4 functions:

In addition, users need to specify the geometry information of the problem. The parameters include the dimension of the grid, the number of nodes (**SIZE**) in each dimension, the length (**LEN**) of each dimension, and if the grid is ring or line topology. If the grid uses ring topology, then the length of each interval between neighboring nodes, for example ΔX , is equal to LEN/SIZE , where **LEN** is the length of X dimension and **SIZE** is the number of nodes in X dimension. Time interval (**Dt**) is also needed as a parameter to compute the new value. Users can change those parameters to experiment different partition of the space in the hope to find the best partition.

5.2. Problem Specification: Submission Window

File Location: /home/hkuang/webMess/apps/FD/tsp/seq/revised/

C File Name: tsp.c

Header File Name: tspm.h

Geometry Dimension Expected in Code: 1 2 3

-X Axis- Element Count: Length: Line Ring

-Y Axis- Element Count: Length: Line Ring

-Z Axis- Element Count: Length: Line Ring

Delta T (Difference in Time Axis):

Stride (How far from neighbor expected in the code): 1 2

Data Initialization Function:

Computation Function:

Output Function:

Termination Condition:

Fixed Number of Steps

L1 Tolerance:

L2 Tolerance:

L Infinity Tolerance:

Number of Machines:

For First Dimension:

For Second Dimension:

For Third Dimension:

Next Cancel Help

Figure 15. Submission window for Finite-Difference paradigm

5.2.1. File Location [Text Field]

The user needs to input the path (FTP format) where they store their original Finite Difference program, so that this system can locate and find those files through FTP. User also needs to make files and directory public so that this system can use anonymous login. However, all the files of the sequential Finite Difference program

should be placed under one single directory. Error will happen if the files are placed under more than one directory.

5.2.2. C File Name [Text Field]

The user needs to input the name of the .c file, which contains the main procedure of the original Finite Difference code. The C program file(s) are assumed to be stored in the directory PODC gets from File Location (5.2.1), and therefore the path names are not required to be specified.

5.2.3. Header File Name [Text Field]

The user needs to input the name of the .h file (header file) of the original Finite Difference program. There should be only one .h file (header file) existing in the program. If not, the user has to combine all the .h files (header files) into one. Error will happen if more than one .h file (header file) is used.

5.2.4. Geometry Dimension Expected in Code

The user has to specify how many dimensions this Finite Difference problem will address (from one to three). The user needs to input this for the convenience of PODC server's parallelization process. However, the number has to match with the parameter in the sequential code, otherwise error will occur.

1 [Radio Button]: If the user chooses one dimension, then the line of X-Axis (5.2.5) and Number of Machines for First Dimension (5.2.14) will be enabled for the user to input.

2 [Radio Button]: If the user chooses two dimensions, then the line of X-Axis (5.2.5), Y-Axis (5.2.6), Number of Machines for First Dimension (5.2.14), and Second Dimension (5.2.14) will be enabled for the user to input.

3 [Radio Button]: If the user chooses three dimensions, then the line of X-Axis (5.2.5), Y-Axis (5.2.6), Z-Axis (5.2.7), Number of Machines for First Dimension (5.2.14), Second Dimension (5.2.14), and Third Dimension (5.2.14) will be enabled for the user to input.

5.2.5. X Axis

Element Count [Text Field]: The user needs to input the number of elements in X axis.

Length [Text Field]: The user needs to input the total length of X axis.

Line or Ring [Radio Button]: The user needs to specify the network topology he/she wants to use.

5.2.6. Y Axis

Element Count [Text Field]: The user needs to input the number of elements in Y axis.

Length [Text Field]: The user needs to input the total length of Y axis.

Line or Ring [Radio Button]: The user needs to specify the network topology he/she wants to use.

5.2.7. Z Axis

Element Count [Text Field]: The user needs to input the number of elements in Z axis.

Length [Text Field]: The user needs to input the total length of Z axis.

Line or Ring [Radio Button]: The user needs to specify the network topology he/she wants to use.

5.2.8. Delta T (Difference in Time Axis) [Text Field]

The user needs to input the distance between each time stamp in time axis (usually is called Δt where t means time). This value will be used to in the computation function (5.2.11) to compute the new value in each dimension of next time stamp.

5.2.9. Stride (How far from neighbor expected in code) [Text Field]

The user needs to input the distance between each node on the direction of X axis. The user needs to input this for the convenience of PODC server's parallelization process. However, the number has to match with the parameter in the sequential code, otherwise error will occur.

5.2.10. Data Initialization Function [Text Field]

The user needs to input the name of Data Initialization Function that generates all the input data, which includes the data from each dimension. This function should only cover one single element in a dimension at a time, so this function should be put inside a loop to calculate all the elements in a dimension. If there is more than one dimension, then input value and return type should be arrays.

Prototype: **Array(s) of Dimension(s) Init(global index of Array(s))**

Input: Index of element in dimension(s) array(s).

Output: Element of dimension(s) array(s).

5.2.11. Computation Function [Text Field]

The user needs to input the name of Computation function, which is the main computation procedure of the sequential Finite Difference program (It is not the main function in standard C). Given its own value and its neighbors' values in the previous time step, it will produce the new value for the current time step.

Prototype: newVals[X] = **Compute**(X, oldVals, Dt, ΔX);

Input: The current value, neighbors' values in previous time step, Δt (5.2.8) and ΔX (Element Count divide by Length in X, Y, or Z axis).

Output: The new values of the node in a dimension.

5.2.12. Output Function [Text Field]

The user needs to input the name of output function, which saves each result from each dimension calculated from the program. The assumption for this function is that the output file is open and ready for write.

Prototype: **WriteResult(outFile, X, oldVals);**

Input: One solution (current one and old one) in one dimension (as a result this function should be put into one loop in order to finish one dimension), and a pointer to an output file, which records all the results.

Output: None.

5.2.13. Termination Condition

The user needs to input the termination condition of this finite difference program. PODC provides four options, which will be described below. The user can only choose one of them, and when one is selected, others will be disabled.

Fixed Number of Step [Radio Button + Text Field]: The finite difference program terminates after a fixed number of iteration. The user also has to input the amount of this number.

L1 [Radio Button + Text Field]: The finite difference program terminates when the tolerance calculated by L1 formula ($\int |f \bullet g|$) is under the user's input of text field in this line.

L2 [Radio Button + Text Field]: The finite difference program terminates when the tolerance calculated by L2 formula ($\sqrt{(\int |f \bullet g|)^2}$) is under the user's input of text field in this line.

L Infinity [Radio Button + Text Field]: The finite difference program terminates when the tolerance calculated by L^∞ formula ($\max |f \bullet g|$) is under the user's input of text field in this line.

5.2.14. Number of Machines

For First Dimension [Text Field]:

For Second Dimension [Text Field]:

For Third Dimension [Text Field]:

These fields will be enabled according to the value of Geometry Dimension Expected in Code (5.2.4) and once enabled, they are optional for the user. If a user knows the number of machines, which can optimize the program, then he/she can input the data. Otherwise the user can just leave it blank to let the system decides the optimized number of machines for the program.

5.3. The Feedback Window

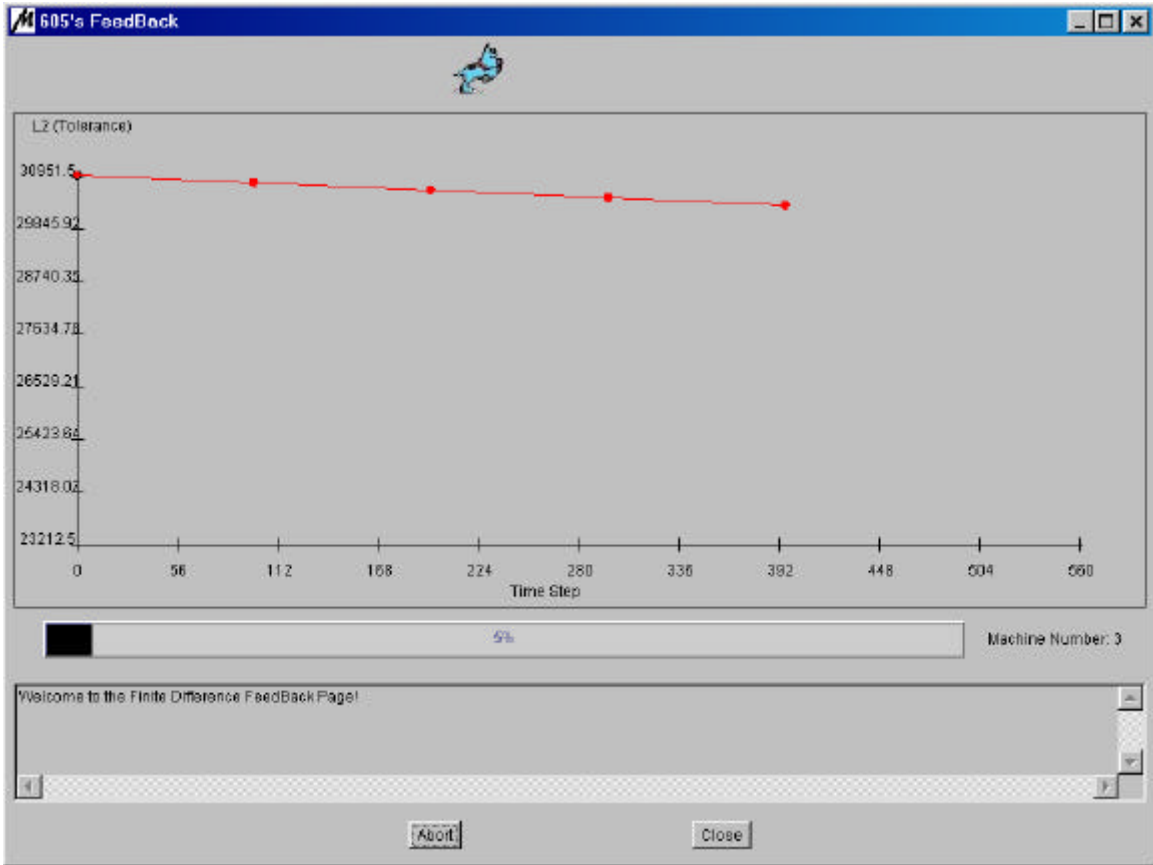


Figure 16. Feedback window for Finite-Difference paradigm

While the application is running, the user and the running application are able to interact. Figure 16 shows the feedback window of finite difference application, which communicates the running status of the application to the user.

At the very top of the window, a moving icon will indicate that the application is currently running. The middle graph is showing the changes of tolerance (Y axis) when the time step (X axis) goes by. There is a small text showing what kind of tolerance is depending on the input of Termination Condition (5.2.13). However, the option of “Fixed Number of Step” displays as the option of “L2”. Below that is the text that will

describe error or success messages. The window also shows the number of machines currently being used by the distributed application.

Chapter 6.

Development Tool

The interface program is completely developed with SUN's original Java Developer Kit version 1.2. It includes the Java interpreter, Java classes and the Java development tools. There are several reasons why Java was chosen as the programming language for this graphical user interface. First, and the most important reason, is Java has the ability to produce graphical objects with very good visual effects yet maintain a reasonable degree of programming simplicity. In other words, Java is a fairly simple programming language that can code graphical materials, unlike APIs in C/C++, and the output from Java language is good enough to interact with the user although the visual effects from Java may not be the best available.

Second, Java is a programming language that is designed to run on the Internet. Platform independence is Java's most special feature. It provides lots of portability to its software and this advantage can be interpreted as efficiency and convenience. Programmer only needs to write the code once and the product can run on different platform means efficiency. In addition, users from different platforms can use the same software without modifying their own machine is convenient. PODC is designed to be a practical tool for computation, and its interface should be done with the purpose of helping PODC to be accepted and used by more people.

Another reason why Java is a best choice for coding the interface design is its extensibility and reusability. PODC currently can only handle four different paradigms, but there may be more paradigms adding to PODC later. Because of the pure object-oriented characteristic of Java, it will be very easy to add another interface for a new

paradigm into the current main interface. In addition, there are lots of components that are used repeatedly in current four different interfaces. It is very simple to inherent (reuse) an existing code segment with the fantastic hierarchy structure of Java.

Chapter 7.

Related Research

This thesis is about a graphical user interface for a paradigm-oriented distributed computation. Paradigm-oriented approach has been popularly employed to build parallel computation models where programmers do not have to be aware of parallelism at all. One example of such approach is an algorithmic skeleton approach to control parallelism that was developed by Murray Cole. Other related work includes Rabhi's that describes some of the common parallel programming paradigms, which explain the basic principles behind a paradigm-oriented programming approach. Gortatch studies extensively the parallel implementation of divide-and-conquer paradigm and its application to the FFT computation.

As to the user graphical interface, there are many of them running for almost any kind of software. However, no one has ever combined these two parts together before, which allows the user to specify application-specific components via a graphical interface. This graphical user interface helps eliminating the difficulty in specifying an application using a not commonly used language because PODC supports C language.

Chapter 8.

Summary and Conclusion

A graphical user interface is usually introduced to simplify the input and output of the program. The main purpose is to allow the users feel comfortable using the software. This interface of PODC has the same goal because of the difficulty of approaching distributed computing. This interface provides an easier way for those people who want to use PODC to speedup their sequential program, and it plays an important role for PODC too since nobody will use a software that is difficult to use. An actual experience proves the importance of this interface. This instance involved solving a computational molecular biology problem. One of the artificial intelligence (AI) research groups in University of California, Irvine, has an application to compute the distance $dist(S1, S2)$ between every pair of strings S1 and S2 stored in a file comprising 6225 strings, and the length of each string is 500 bytes. The distance is computed using the Smith and Waterman algorithm, which takes approximately 1.16 second for each pair of strings when the application is designed in sequential mode. The group felt that it takes too much time to finish the job, which is approximately a month. They heard that PODC probably could help this application to achieve some speedup. In addition, this application was also a perfect candidate for distributed execution using the bag-of-tasks paradigm because of the independent nature of this computation. It took two hours for the user to understand how PODC parallelizes the problems of the bag-of-tasks paradigm, and three hours to restructure the sequential program so that it fit the requirements of the interface. The result was very pleasing to the group, which took 2.4 days using 16 machines and had a speedup of 15.1. The graphical user interface played an important

role here because without this interface, the user could have spent several days to communicate with the person who operates PODC or even months to learn PODC and rewrite the program by himself.

However, a graphical user interface is not always perfect. There are always certain tradeoffs between the generality and ability to completely solve the problem. If the interface is too general, then it can fit many types of problems, but it will not be able to go too deep for each type of problem. In addition, if the interface can help to solve one type of problem perfectly, then this interface definitely cannot be used for the interface of other kinds of problems. The graphical user interface for the PODC is trying to compromise between these two opposite directions.

This graphical user interface reduces the complexities of the scaling process by providing default values to input control whenever possible, presenting a list of choices when appropriate, and displaying the extensive statistical output in table and graph form for easier viewing.

Chapter 9.

Future work and further development

Currently this graphical user interface is written in application style, which means the user needs to download the source code or classes of this interface from web and run it under the console environment of the local machine. One of the improvements that can be done in the future is allow the users to use PODC through the web. In fact, another web version of this interface has been attempted. This web version has been stopped because of certain security issues involving the fact that Java does not allow PODC server to directly access some resources of local machine to draw part of the diagram of the interface. Nevertheless, putting the services of PODC on the web is still a practical goal. It will be an easy and efficient way to utilize PODC because the user can just access the web page to use PODC. If the local browser does not support the Java version the interface uses, the user just needs to download latest version of Java Plug-In (JPI) software and Java Runtime Environment (JRE) instead of downloading the entire package for running the application on the local machine. The JPI will enable the web browser to use the JRE to run applets.

Another improvement that can be done in the future is adding more paradigms into this interface so that this interface can handle more problems. However, this suggestion also needs the cooperation from the core – PODC server. The graphical user interface between users and server can only be useful when the server can parallelize that type of problem.

BIBLIOGRAPHY

- Lubomir F. Bic, Michael B. Dillencourt, and Munehiro Fukuda. *Mobile Agents, DSM, Coordination, and Autonomous Object Paradigms: A Common Framework*. 1997. Department of Information and Computer Science, University of California Irvine.
- Lubomir. F. Bic, Michael. Dillencourt, Munehiro Fukuda. Aug. *Distributed computing using autonomous objects*. 1996. IEEE Computer.
- Lubomir F. Bic, Michael B. Dillencourt, Munehiro Fukuda. *Messages versus messengers in distributed programming*. 1999. Journal of Parallel and Distributed Computing.
- Lubomir F. Bic, Michael B. Dillencourt, Munehiro Fukuda. 1998. *Distributed coordination with messengers*. 1998. Science of Computer Programming.
- Adamidis, P. *Review of parallel genetic algorithms bibliography*. 1994. Thessaloniki, Greece: Aristotle University of Thessaloniki.
- Cant'u-Paz, E. *Designing efficient master slave parallel genetic algorithms*. 1997. Urbana, IL: University of Illinois at Urbana Champaign.
- Lin, S.-C., Punch, W., & Goodman, E. *Coarse grain parallel genetic algorithms: Categorization and new approach*. In *Sixth IEEE Symposium on Parallel and Distributed Processing*. 1991. IEEE Computer Society Press.
- Goldberg, D. E., & Deb, K. *A comparative analysis of selection schemes used in genetic algorithms*. 1991. Foundations of Genetic Algorithms.
- Goldberg, D. E. *Genetic and evolutionary algorithms come of age*. 1994. Communications of the ACM.
- Goldberg, D. E., Deb, K., & Clark, J. H. *Genetic algorithms, noise, and the sizing of populations*. 1992. Complex Systems.
- Holland, J. H. *A universal computer capable of executing an arbitrary number of sub-programs simultaneously*. 1959. Proceedings of the 1959 Eastern Joint Computer Conference.
- Manderick, B., and Spiessens, P. *Fine-Grained Parallel Genetic Algorithms, in Proc. of the Third Int. Conf. on Genetic Algorithms*. 1989. J. D. Schaffer (Editor), Morgan Kaufman.
- Starkweather, T., Whitley, D., and Mathias, K. *Optimization Using Distributed Genetic Algorithms, in Parallel Problem Solving from Nature, Lecture Notes in Computer Science*. 1991. Schwefel and R. M'anner (Editors), Springer-Verlag.

Kairong Kuang, Lubomir F. Bic, Michael B. Dillencourt. *Technical Report*. 1999. Department of Information and Computer Science, University of California Irvine,

Thomas A. Standish. *Data Structures in Java*. 1998. Addison Wesley Longman, Inc.

Tomassini, M. *The Parallel Genetic Cellular Automata: Application to Global Function Optimization, Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*. 1993. Springer-Verlag, Wien.

Muhlenbein, H., Schomish, M., and Born, J. *The Parallel Genetic Algorithm as Function Optimizer*. 1991. Parallel Comput.

Matthew T. Nelson. *JAVA Foundation Classes*. 1998. The McGraw-Hill Companies, Inc.

Patrick Niemeyer, and Joshua Peck. *Exploring Java, Second Edition*. 1997. O'Reilly & Associates, Inc.

Bruce Eckel. *Thinking In Java*. 1998. Prentice Hall PTR Prentice-Hall Inc.