Integrating Task Parallelism in Data Parallel Platforms for NOWs

Binu K J, D Janaki Ram

Distributed and Object Systems Group

Department of Computer Science and Engineering

Indian Institute of Technology Madras, Chennai - 600 036

India

Email: binu@lotus.iitm.ernet.in, djram@lotus.iitm.ernet.in

Abstract

A number of high level parallel programming platforms for Network of Workstations (*NOWs*) have been developed in the recent times. Most of these platforms target to exploit data parallelism in applications. They do not allow expressibility of applications as a collection of tasks along with their precedence relationships. As a result, the control or task parallelism in an application cannot be expressed or exploited. The current work aims at integrating the notion of task parallelism and precedence relationships among constituting tasks to such high level data parallel platforms for NOWs. Our model of integration provides for arbitrary nesting of data and task parallel modules. Also, the precedence relationships are clearly reflected from the program structure. The model relieves the programmer from handling non-determinism with respect to the order of completion of tasks. Design of the runtime support is also discussed. The model is general enough to be integrated to a wide range of data parallel platforms. A specific case of integrating the model into Anonymous Remote Computing(ARC), a data parallel programming platform for loaded heterogeneous workstations is presented. The performance analysis shows the significance of exploiting both task and data parallelism present in a problem. Also, we show that by exploiting both task and data parallelism, the optimum number of nodes for parallel execution of a problem can be increased.

Keywords: Parallel Programming; Data Parallelism; Task Parallelism; Network of Workstations; Loosely Coupled Distributed Systems; Distributed Problem Solving.

1 Introduction and Motivation

Data Parallelism refers to simultaneous execution of same instruction stream on different data elements. Several programming platforms target to exploit data parallelism [1][2]. Control parallelism refers to the simultaneous execution of different instruction streams[2]. This is also referred to as task parallelism or functional parallelism[2]. Some of the tasks that constitute the problem may have to honor precedence relationships amongst themselves. Task parallelism with precedence constraints can be expressed as a task graph where nodes represent tasks and directed edges represent their precedences. It is the parallel execution of distinct computational phases that exploit a problem's task parallelism[3]. Task parallelism is important for various reasons. Some of these are discussed below.

- *Multidisciplinary applications*: There is an increased interest in parallel multidisciplinary applications where different modules represent different scientific disciplines. These modules may be implemented for parallel computation[4]. The airshed model[5] is an example. It is a grand challenge application which characterizes the formation of air pollution as the interaction between wind and reactions among various chemical species.

- *Complex simulations*: Most of the complex simulations developed by scientists and engineers have potential task and data parallelism[6]. A data parallel platform would not be able to exploit the potential task parallelism inherent in such problems.

- *Software engineering*: Independent of issues relating to parallel computing, treating separate programs as independent tasks may achieve benefits of modularity[7].

- *Real time requirements*: Real-time applications are characterized by their strict latency time and throughput requirements. Task parallelism lets the programmer explicitly partition resources among the application modules to meet such requirements[4].

- *Performance*: Task parallelism allows the programmer to enhance locality and hence performance by executing different components of a problem concurrently on disjoint sets of nodes. Also it allows the programmer to specify computation schedules that could not be easily discovered by a compiler[7].

- *Problem characteristics*: Many problems can benefit from a mixed approach. For instance, a task parallel coordination layer can integrate multiple data parallel computations. Some problems admit both data and task parallel solutions, with the better solution depending on machine characteristics[7].

Very often, task and data parallelism are complementary rather than competing programming models. Many problems exhibit a certain amount of both data parallelism and task parallelism. It is desirable for a parallel program to exploit both data and task parallelism inherent in a problem. Hence, a parallel programming environment should provide adequate support for both data and task parallelism[8]. The current work aims at integrating task parallelism into data parallel platforms for network of workstations.

2 A model for integrating task parallelism into data parallel programming platforms

2.1 Expectations from an integrated platform

The expectations from a high level parallel programming platform stem from the nature of applications which could utilize the platform. The requirements come from the desired expressibility of the application, possible transparency in programming, exploitation of parallelism, achievable performance optimizations for the application etc. These are explained below.

- *Expressibility*: In order to exploit parallelism in an application, the program must express potential parallel execution units. The precedence relationships among them also must be expressed in the program. An elegant expressibility scheme should reflect the task parallel units, data parallel units and precedence dependence among the tasks in the program. This would ease programming, improve readability and enhance maintainability of the code. However, the expressibility that can be provided is influenced by the nature and organization of the underlying runtime support and the native language to which it is converted.

- *Transparency*: It is desirable to relieve the programmer from details relating to underlying network programming. This results in the programmer concentrating on his application domain itself. With network programming details coded in the application, a major portion of the program will be unrelated to the application. Consequently, such programs suffer from readability and hence maintainability.

- *Performance*: System level optimizations by the parallel programming platform can improve performance of applications. In addition, the system can achieve load balancing for the application, further enhancing performance. The run time scheduling decisions by the system, both the time of scheduling and node to be scheduled are the other factors that can improve performance.

Other desirable properties of the system include fault resilience, fault tolerance, accounting for heterogeneity in machine architecture and operating system and portability of application.

2.2 Programming model

The model aims at a parallel programming platform that permits expressibility of task and data parallelism so that both could be exploited. In view of a large number of existing data parallel programming platforms for NOWs, it would be useful to formulate the problem as integrating task parallelism into existing data parallel platforms.

Two issues have been considered while integrating task parallelism into data parallel platforms. The first issue relates to expressibility of task parallelism in existing data parallel platforms. The data parallel model of computation of existing platforms would be different. Consequently, the program structure favored by these platforms would also be different. Hence, at the programming level, the model restricts itself to expressing tasks and their precedence relationships. Our approach ensures a seamless integration of task parallelism into existing data parallel platforms.

The model permits a block structured specification of the parallel program with arbitrary nesting of task and data parallel modules. This is one of the most important quality of an integrated model[8]. A block could be a high level characterization of a data parallel module in accordance with the principles of the underlying data parallel platform. This expressibility reflects the task parallel blocks and precedence relationships in the task graph.

The second issue pertains to data parallel subdivision of tasks. The underlying data parallel model may subdivide a data parallel task into subtasks. These subtasks could be migrated to different nodes over the network. The completion of a task could be a significant event in the proposed integrated model since tasks have precedence relationships.

In the model, the system takes the responsibility to intimate the user process when an event of its interest occurs. An events of interest signify completion of one or more tasks which meet the precedence requirements for another task. Hence, the model takes care of the probability factor in the order of completion of tasks that constitute a program. The model provides templates by which a user program can register the events of its interest with the system.

Blocks of code are demarcated by the constructs viz. *Task_begin* and *Task_end*. Another construct, viz. *OnFinish* is provided to specify the precedence requirements of a task. The syntax and semantics of these constructs are described below.

```
Task_begin(char *TaskName) OnFinish(char *WaitforTask, {AND,OR} ...)

This marks the beginning of a block. A block signifies a task. The name
of the task it signifies is furnished along with "Task_begin". If the
task has precedence relationships to be met, the "Task_begin" is followed
by the construct "OnFinish" with the names of tasks it has to wait for
as its arguments. If "OnFinish" is not furnished, it will be presumed
that there are no preconditions to the task. "OnFinish" can take a
variable length list of arguments. The individual tasks which are
arguments to OnFinish could have "AND" or "OR" relationship between them.
"AND" and "OR" have the same precedence. Their associativity is defined
as left to right. Notation adopted for "AND" and "OR" are "&" and "|"
respectively.

Task_end(char *TaskName)

This marks the end of a block. The name of the task it signifies is
furnished along with "Task_end".
```

2.3 Program Structure and Translation of a task graph

Figure 1 presents a sample task graph to illustrate the expressibility provided by the model. The corresponding block structured code is given in Pseudo Code 1. It illustrates the translation of a given task graph into the program structure favored by the model.

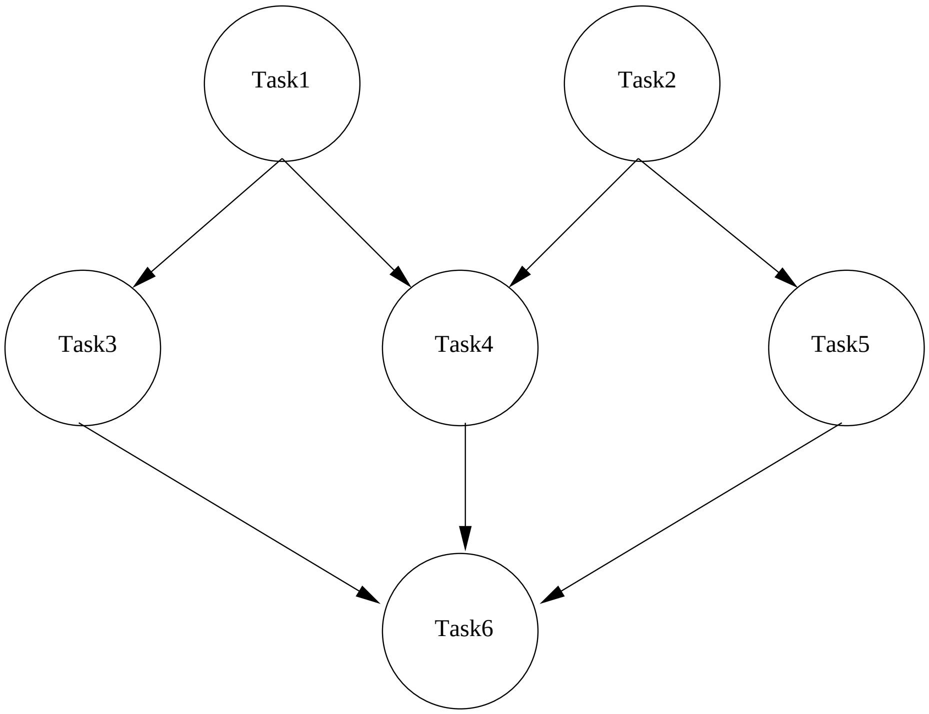

Figure 1: **A sample task graph with precedence relationships**

The corresponding Pseudo Code is given below.

```
Pseudo Code 1 : The block structuring corresponding to the task graph in
Figure 1.

Task_begin(Task1)
        ...
        Task_begin(Task3) OnFinish(Task1)
                ...
        Task_end(Task3)

Task_end(Task1)

Task_begin(Task2)
        ...

        Task_begin(Task4)  OnFinish(Task1 & Task2)
```

```
        ...
    Task_end(Task4)

    Task_begin(Task5) OnFinish(Task2)
        ...

        Task_begin(Task6) OnFinish(Task3 & Task4 & Task5)
            ...
        Task_end(Task6)

    Task_end(Task5)

Task_end(Task2)
```

The program in Pseudo Code 1 shows the expressibility of a task graph in the model. The two outer blocks, *task1* and *task2* signify the tasks which could be executed task parallely at the beginning of the run itself. They have no precedence relationship to be met. Hence, their *Task_begin* constructs are not followed by *OnFinish* directives.

A task which has precedence relationship with another task is written inside its predecessor task's block. Also, the *Task_begin* construct of such a task would be followed by the construct *OnFinish* which specifies the preconditions. This could be observed from the specification of *task3*. *task3* is placed inside the block of of its predecessor task viz. *task1*. Also, it could be noted that the *OnFinish* construct corresponding to the block of *task3* declares its precondition.

When the precondition of a task comprise of completion of more than one task, it can be placed inside one of those blocks which represents a predecessor task. However, its *OnFinish* specification should list the names of all its immediate predecessors. This could be observed from the specification of *task4* and *task6*.

2.4 Separation of System's and Programmer's concern

Our model attempts to make a clear separation between programmer's concern and system's concern. At the programmer's level, expressibility is the main concern, whereas at the system's level issues of non-determinism and performance are the major concerns.

Arguments from the expressibility point of view favor a control structure that best reflects the precedence relationships in the code. This in turn could argue for structuring of programs such that the pieces of code that are executed at the completion of its predecessor task appear as a program segment inside the program segment of (one of) its predecessor task itself. Such a program structure suitably describes the task graph and contributes to the elegance of the code. However, it does not support non-determinism in the order of completion of tasks according to the flow of control permitted by traditional languages.

A flat control structure like *switch-case* could be utilized to support non-determinism in the order of completion of tasks. However, it would lead to a poor program structuring. Hence, to alley the differences between these demands, the program translator allows a program expressibility scheme which satisfies the expectations from the expressibility point of view. This in turn is parsed and translated to a control structure of the native language that well answers the concerns of non determinism at the system's level.

The program translation provided with the current model permits a control structure similar to the block structured code that programmers are familiar with. At the same time, it reflects the task graph and precedence relationships of the application.

3 Integration of the model into ARC

3.1 ARC model of computation

In Anonymous Remote Computing(ARC) Paradigm[9], a parallel program for NOW is written as a collection of several loosely coupled blocks called *Remote Instruction Blocks* (RIB) within a single program entity. An RIB is a code fragment that can be migrated to a convenient anonymous remote node at run time for execution. RIBs do not involve any mechanism of process creation or inter-task communication at the programming language level. The nodes at which RIBs are to be executed remains anonymous to the program. The ARC runtime system decides the nodes for RIB execution. At a given time, multiple programs could be generating RIBs and multiple anonymous remote participants could be joining or leaving the ARC system. ARC addresses heterogeneity in architecture and operating system, fault tolerance, load balancing with other load coexisting and resilience to changing availability of nodes. However, ARC targets data parallel applications. The task parallelism along with their precedence relationships cannot be expressed in ARC.

In order to achieve load balancing, the ARC model provides a system service which can be availed by the user process to get the current availability and load of machines. The user process can use this information to migrate the RIBs to least loaded machines. For data parallel modules, the problem can be divided into appropriate grain sizes depending on the load of the machines. Subsequently, each grain of computation can be migrated to corresponding machines. The ARC model provides two function calls, one to get the status of a submitted task and the other to collect the results. The syntax and semantics of the calls supported by ARC are given below.

```
LFmessage get_load_factor(int numberOfMachinesNeeded)
```

```
This call is used to obtain information about the machines available for
parallel computation and their loads.  The argument to this call is the
number of machines required. The return value is a structure which gives
the number of machines actually available, their load information, and
machine indices which are used by the underlying system to identify the
machine.
```

```
ARC_function_call(char *funct_name, int timeout, int retries, char *
arg_data, int arg_size, int result_size, int machine_index, int tag)
```

```
This call is used to submit a task. The first argument specifies the
function that specifies the task. The second argument is the timeout
period. The third argument is the number of retries that should be made.
The fourth argument is the raw data that represents the arguments to the
task. The fifth argument is the size of the result. The next argument is
the index of the machine, obtained from "get_load_factor" call. The last
argument is the tag to identify the particular task submitted. The value
of tag is returned by the call.
```

```
char *obtain_results(int tag);
```

```
This call is to collect results of a task. The only argument is the
unique id of the task. The call blocks till results are available.
```

```
int peek_results(int tag);
```

```
This is a non blocking call for a program to check if the results are
available. The only argument is a tag for the piece of computation of
which result is requested. Returns 1 if the results are available.
Else it returns -1. When the results are available, it can be collected
by obtain_results call.
```

3.2 Outline of ARC runtime support

The runtime support of ARC[10] consists of a local co-ordinator (lc) daemon running on each machine participating in parallel computation and a system co-ordinator (sc) daemon on a designated machine.

The *lc* co-ordinates the user processes initiated on the machine on which it runs. A process can get itself registered with the *lc* by invoking the call viz. *initialize_ARC*. Complementing the call is *close_ARC* to de-registers a user process from *lc*. The syntax and semantics of the calls are given below.

```
void initialize_ARC(void);

It takes no arguments and returns no values. It registers the user process
with the runtime system. In response to this call, the "lc" allots an
exclusive communication channel between the "lc" and the user process for
subsequent communication.

void close_ARC(void);

It takes no arguments and returns no values. It de-registers the user
process from the runtime system. The system updates its tables accordingly.
```

The *sc* co-ordinates the *lcs* in the pool and hence the machines participating in the pool. *sc* keeps track of individual *lcs* and facilitates communication between them. Also, the *sc* provides information regarding availability and load on machines.

The *lcs* and *sc* communicate by predefined messages (through a dedicated channel). Similarly, communication between a user process and *lc* is also by predefined messages. Typically, a message sequence is initiated by a user process by sending a message to its *lc*. The *lc* either replies to the message or generates a message to the *sc*. The *sc* gathers the relevant data and returns to the *lc*. Subsequently, the *lc* replies to the user process which initiated the message sequence.

3.3 The Integrated Platform

The integration of task parallelism into the ARC framework includes providing for additional function calls and modifications to the existing runtime support. These additional calls include calls to initialize and close the system, a call to register a task as a collection of subtasks and a call to register events of interest with the system,

A user process registers with the system by invoking the call viz. *TaskInit()*. Complementing the call is *TaskClose()* which de-registers a user process from the system.

The ARC model permits division of a task at run-time depending on the availability and load on the machines. This permits the decision of number of subtasks for a task to be deferred till run-time. The function call viz. *task()* facilitates registering a task as a collection of subtasks. The function can be invoked at run-time after the division of task. The call permits a variable length argument list to accommodate varying number of subtasks.

The user process registers the events of interest with the system using the call viz. *RegisterEvent()*. The call is inserted by the translator during program translation.

The syntax and semantics of the calls are given below.

```
int task(char *TaskName, int SubTaskId, ...)

The call registers a task as a collection of its subtasks. The first
argument is the name of the task. The second argument is a subtask id.
If the task is not subdivided, the task_no itself is furnished here.
If the task is subdivided, there would be more arguments each of which
representing a subtask. The number of arguments depends on the the
number of subtasks that constitute the task. The return value is either
SUCCESS or ERRNO corresponding to the error.

int RegisterEvent(int EventId, char* TaskNames, ...)
```

The call registers an event of interest with the system. The first argument is the event id. Following it is a variable length argument list of task names constituting the event. The return value is either SUCCESS or ERRNO corresponding to the error.

In addition to the functionalities related to data parallelism, the run-time system handles mapping tasks to their subtasks, storing the events of interest for a user process and keeping track of the execution status of tasks. Message protocols are defined to access and modify these tables. The integrated platform uses these information to intimate a user process, when an event of it's interest occurs. The intimation to the user process is by a predefined message.

3.4 A sample block in the integrated platform

The program structure of the integrated platform was shown in Pseudo Code 1. This section presents a block in the program. As mentioned earlier, the code inside a block would be written using ARC calls given in section 3.1. A typical block in a program is given below (Pseudo Code 2).

Pseudo Code 2: A typical block in the integrated platform.

```
Task_begin(TaskN) OnFinish(TaskM)

        // Collect the results of an earlier task for subsequent processing.
        // The earlier task was data parallelized into two parts with
        // TaskMTag1 and TaskMTag2 representing subtasks.

        ObtainResults(TaskMTag1);
        ObtainResults(TaskMTag2);

        // GetLoadFactor returns a structure which gives details of the

        // available machines, their loads, processing powers etc.
        // The only argument specifies the maximum number of machines
        // sought for.
```

```
    MachineAvailability =  GetLoadFactor(3);

    if (MachineAvailability.Count == 3)
            {
            // Data parallelize into 3 with the division based on the
            // load and processing power of three machines returned.
            ARC_function_call(TaskN, ..., TaskNTag1);
            ARC_function_call(TaskN, ..., TaskNTag2);
            ARC_function_call(TaskN, ..., TaskNTag3);
            task(TaskN, TaskNTag1, TaskNTag2, TaskNTag3);
            }

    if (MachineAvailability.Count == 2)
            {
            // Data parallelize into 2 with the division based on the
            // load and processing power of three machines returned
            ARC_function_call(TaskN, ..., TaskNTag1);
            ARC_function_call(TaskN, ..., TaskNTag2);
            task(TaskN, TaskNTag1, TaskNTag2);
            }

    if (MachineAvailability.Count == 1)
            {
            // Cannot be data parallelized due to non availability
            // of nodes. Hence run as a sequential program.
            ARC_function_call(TaskN, ..., TaskNTag1);
            task(TaskN, TaskNTag1);
            }

Task_end(TaskN)
```

The block is demarcated by *Task_begin* and *Task_end*. The *OnFinish* directive specifies the completion of *TaskM* as the precondition of *TaskN*. *ObtainResults()* call is used to obtain results of an earlier task.

It could be seen from the code that the number of data parallel subdivision of *TaskN* is decided at the runtime. Hence, the *task()* call to register the task as a collection of subtasks has to be invoked accordingly. The code also shows how the *task()* call is utilized to register a task as a collection of subtasks at the runtime.

4 Design and Implementation

4.1 Program Translator

The program translator translates the proposed task parallel model into the appropriate execution model. This involves detection of tasks having precedence constraints and generation of appropriate code to handle their execution at run-time. The translator also generates the network related code.

In the first pass, the translator identifies tasks and their precedence constraints. Tasks without any precedence constraints are placed at the beginning of the code. For the tasks having precedence constraints, calls to register their constraints as events of interest are generated. When such an event occurs, it triggers the execution of the task waiting for the pre-condition. Tasks having the same set of pre-conditions are grouped under the same event of interest.

The translator constructs code for detecting and handling events of interest in the second pass. The translator constructs an infinite loop to read event interrupts on a socket. When an event occurs, the task waiting for the event will be notified and appropriately scheduled by the run-time system.

A sample translated code is given below (Pseudo Code 3). The code is obtained by translation of Pseudo Code 1. The transformations by the translator could be seen by mapping the following code with Pseudo Code 1.

```
Pseudo Code 3: A translated code for Pseudo Code 1

/* Sample Defines file */
#define EVENT_Task1 1
#define EVENT_Task2 2
#define EVENT_Task1_Task2 3
```

```
#define EVENT_Task3_Task4_Task5 4


/* Register Events */
RegisterEvent(EVENT_Task1, "Task1")
RegisterEvent(EVENT_Task2, "Task2");
RegisterEvent(EVENT_Task1_Task2, "Task1", "Task2");
RegisterEvent(EVENT_Task3_Task4_Task5, "Task3", "Task4", "Task5");

/* Contents of the Block for Task1 */
...
/* Contents of the Block for Task2 */
...

while(1)
{

  Event = WaitForEvent();

  switch (Event)
    {
     case EVENT_Task1 :
       /* Contents of the Block for Task3 */
       ...
       break;

     case EVENT_Task2 :
       /* Contents of the Block for Task5 */
       ...
       break;

     case EVENT_Task1_Task2 :
       /* Contents of the Block for Task4 */
       ...
       break;
```

```
    case EVENT_Task3_Task4_Task5 :
      /* Contents of the Block for Task6 */
      ...
      exit();
      break;

    }
}
```

4.2 Local Co-ordinator

The *lc* runs on each node that participates in parallel computation. *lc* services requests generated by user processes on its node. Also, it maintains relevant information to coordinate the user processes.

In ARC, *lc* maintains three tables, viz.*Program and Task Table (PTT)*, *Recovery Information Table (RIT)*, and *Results List Table (RLT)*. PTT maps the process ids of each process to the socket descriptor that connect it to *lc*. Also, it distinguishes processes as user processes initiated on the node and tasks migrated from other nodes. RIT maintains information relevant for recovery in the event of a failure. RLT stores the results of an earlier submitted tasks when it is available. The results are retained till it is claimed by the corresponding user processes.

The integrated platform maintains two additional tables viz. *Task Table (TT)* and *Event Table (ET)*. These tables are maintained on a per process basis.

TT maps tasks to its subtasks. The table enables the system to keep track of the status of tasks. An entry is created in the table when a task subdivides. The information furnished by the *task()* call is used for the same. Further, the entry is updated when any of its subtasks completes. ET stores the pre-declared events of interest to a process. An entry is created in the table when an event of interest is registered using a *RegisterEvent()* call. Further, it is updated when a task is completed. The structure of ET and TT are given below.

```
Structure of Event Table (ET) and Task Table(TT) :

struct EventTable
```

```
{
int EventIdentifier;         // Event Name
char ** WaitForTaskNames;  // Task Names for this Event
BOOLEAN* TasksOver;          // Task Completed Array
int NumberofWaitForTask;     // Number of Task for the Event
}

struct TaskTable
{
char TaskName[TASKNAME_LENGTH];  // Name of the Task
int * SubTaskIdentifier;         // Sub task identifiers
BOOLEAN * SubTaskOver;           // Sub task completion status
int NumberofSubTask;             // Number of subtasks
}
```

When a migrated subtask finishes its execution, the *lc* is intimated by the *sc*. The *lc* updates the TT and checks if it triggers the completion of a task. If a task is completed, it updates the ET and checks if it satisfies the conditions for any event of interest to the user process. *lc* initiates a message to the user process if any event of its interest occurs. If the completion of a subtask results in more than one event of interest, the user process is intimated by separate messages for each event.

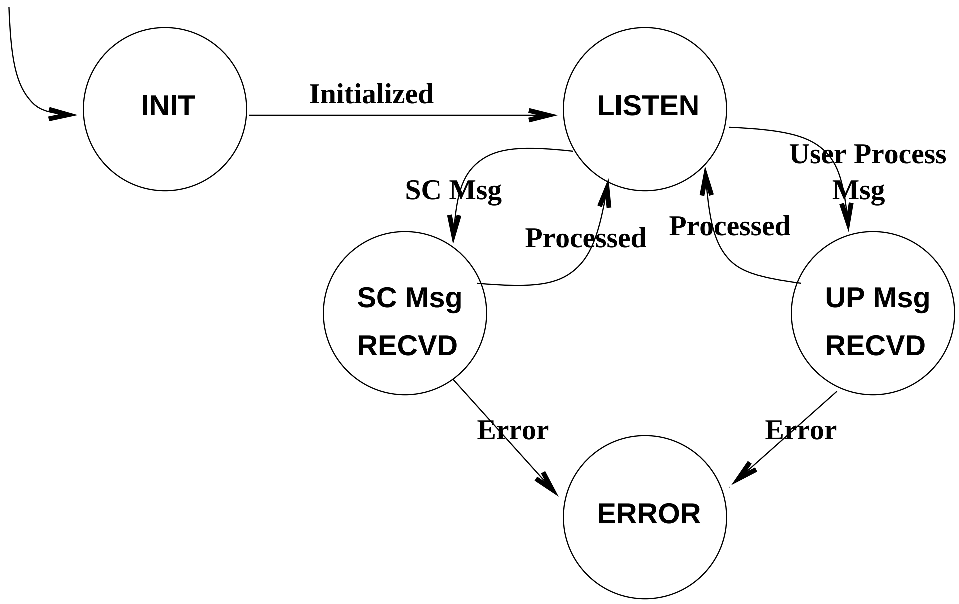

Figure 2: **FSM of LC**

The FSM of *lc* is given in Fig. 2. In the **INIT** state, *lc* initializes its data structures and cleans the auxiliary system files. The *lc*, establishes a TCP connection with *sc* and gets itself registered with *sc*. In **LISTEN** state, the *lc* waits for messages from the *sc* or any user process. On a message from *sc*, it transits to **SC Msg RECVD** and service the message. On a message from a user process, it transits to **UP Msg RECVD** and service the request.

The initial communication between a process and the *lc* is through a known common channel. This is for a user process to get itself registered with the local coordinator. User processes which are registered with the *lc* are given exclusive communication channels for subsequent communication.

4.3 System Co-ordinator

It has been mentioned that *sc* co-ordinates the *lcs* in the pool. Also, the *sc* keeps track of individual *lcs* and facilitates communication between them. The *sc* is additionally responsible for supplying information regarding availability and load of machines.

The *sc* is connected to *lc*s through TCP sockets. *sc* maintains TCP socket descriptors which connect it to individual *lcs*. For small sessions, the *sc* routes the messages between *lc*s so that the overhead for frequent connection establishment and closing is minimized. The message structure includes a field to indicate the destination address in order to facilitate the routing.

The design of ARC has consciously kept the *sc* as thin as possible. Consequently, the *sc* scales up well with respect to the number of *lc*s that participate in the pool.

The FSM of *sc* is given in fig. 3. In the **INIT** state, *sc* initializes its data structures and cleans the auxiliary system files. In **LISTEN** state, the *sc* polls for connection requests from *lcs*. On a connection request from a *lc*, it registers the *lc* with the system and establishes a TCP socket connection between them. Further, it listens for messages from the *lcs* on exclusive channels as well as new connection requests. On a message from an *lc*, it transits to **LC Msg RECVD** state and processes the message. Once the message is processed, it returns to **LISTEN** state.

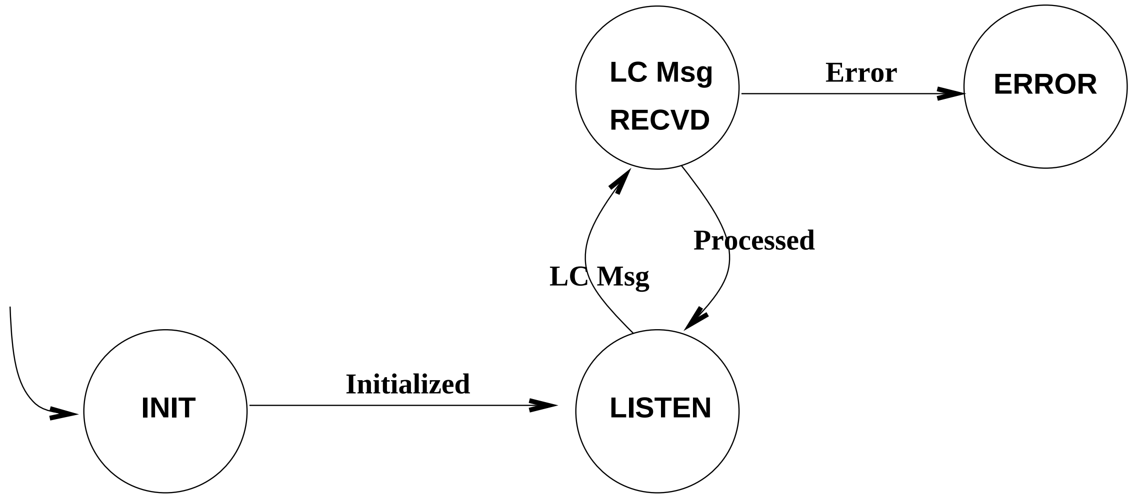

Figure 3: **FSM of SC**

5 Applications

A number of applications have both task and data parallelism in them. Examples of such application domains are speech signal processing, image processing, matrix computations and scheduling algorithms.

A number of signal processing applications consist of signal transformation steps followed by filter steps. The transformation steps could often be done in a data parallel fashion, whereas the filter steps can be executed only after the transformation steps. If there are multiple signal sources, each of them follows the above steps. Such problems can be elegantly expressed in our model. Image processing also displays similar characteristics. Stereo image processing typically involves similar processing of signal from more than one source. The processing steps, in turn, could consist of a sequence of transformations and filtering. Matrix computations performed in many engineering computations such as Finite Element Analysis typically involve processing of huge matrices. Such applications can also utilize the capabilities of our model.

A specific application in the domain of signal processing viz. Speaker verification problem, is discussed below. The task graph for the problem is given in Fig. 4. A sample of the time domain signal is the input to the system. Linear predictive analysis[11] of the signal gives linear predictive (LP) co-efficients. LP Cepstrums, which are features of the input signal are derived from these LP co-efficients. These LP Cepstrums are used for speaker verification. Different methods could be used to obtain evidences for verification. Gaussian Mixture Model(GMM) method[12], Neural Network Method

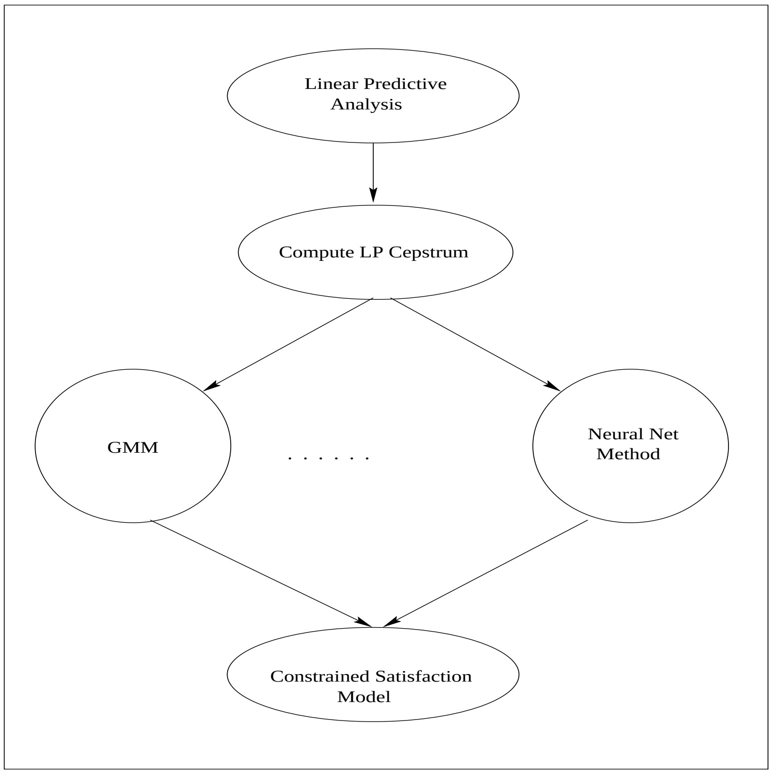

Figure 4: **Task graph of the application**

etc. are some examples. Some methods prove better than the rest according to the nature of input set. Different methods could be applied parallely on the same set of LP Cepstrums. *Constraint Satisfaction Model*[13] combines evidences obtained from each of these models. The data parallelism in each of these methods can also be exploited.

The pseudo-code of the application for the integrated platform is given below (Pseudo Code 4).

```
Pseudo Code 4 : Pseudo Code of the application in fig. 4.

LPAnalyse();
ComputeLPCepstrums();

// GMM and NeuralNet blocks are Task parallelized
Task_begin(GMM) // Start of GMM ARC block
```

```
// migrate to 4 lightly loaded nodes
GetLoadFactor(4);

// Data parallelized
        ARC_function_call(GMM,...,GMMTag1);
        ARC_function_call(GMM,...,GMMTag2);
        ARC_function_call(GMM,...,GMMTag3);
        ARC_function_call(GMM,...,GMMTag4);

Task_end(GMM)

Task_begin(NeuralNet) // Start of NeuralNet ARC block


// migrate to 3 lightly loaded nodes
GetLoadFactor(3);

// Data parallelized
        ARC_function_call(NeuralNet,...,NeuralNetTag1);
        ARC_function_call(NeuralNet,...,NeuralNetTag2);
        ARC_function_call(NeuralNet,...,NeuralNetTag3);

// Wait for GMM and NeuralNet to complete
Task_begin(ConstriantStatisfactionModel) OnFinish(GMM & NeuralNet)

ObtainResult(GMMTag1);
ObtainResult(GMMTag2);
ObtainResult(GMMTag3);
ObtainResult(GMMTag4);

ObtainResult(NeuralTag1);
ObtainResult(NeuralTag2);
ObtainResult(NeuralTag3);

CalculateConstriantStatisfactionModel();
```

```
Task_end(ConstriantStatisfactionModel)

Task_end(NeuralNet)
```

6 Performance Analysis

The performance analysis presented intends to show the advantage of exploiting both task and data parallelism, inefficiency in presupposing the order of completion of tasks and performance advantage in supporting non-determinism in the order of completion of tasks.

The problem considered for performance analysis has two task parallel arms. Each arm consists of two tasks which are amenable to data parallel subdivision. The tasks in each arm have precedence constraints. A problem with more than one task in each arm is chosen to bring out the compensating and cumulating effect in the time of completion of an arm. The task graph of the problem is given in Figure 5. The figure shows the task parallel arms and precedence relationships.

In the problem, Tasks *T1* and *T2* can be executed as the program starts its execution. *T1* and *T2* are matrix multiplications on two independent sets of huge matrixes. Task *T3* is a user defined function on the resultant matrix of *T1*. Hence, *T3* can be executed only after *T1* completes its execution. Similarly, *T4* is a user defined function on the resultant matrix of *T2*. Hence, *T4* can be executed only after *T2* completes its execution. Task *T5* collects the results of *T3* and *T4*.

The arms of the task graph could be represented as:

$$(U(A * B)) \quad and \tag{1}$$

$$(U(C * D)) \tag{2}$$

where A,B,C and D are four square matrices

U	is the user defined function
T1	is A*B
T2	is C*D
T3	is U(A*B)
T4	is U(C*D)

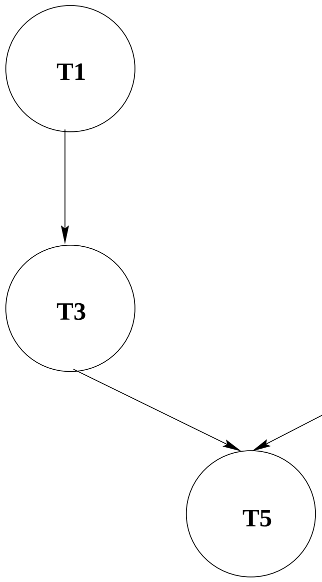

Figure 5: **Task graph of the application**

The two task parallel arms could be a part of any computation. For instance $(U(A * B)) * (U(C * D))$ or $(U(A * B) + (U(C * D))$. A number of other problems also could be represented by the same task graph. Any matrix computation of the form

$$((A \ bop \ B) \ bop \ C) \ bop \ ((D \ bop \ E) \ bop \ F) \qquad or \tag{3}$$

$$((A \ bop \ B) \ bop \ C) \ bop \ (uop \ (C \ bop \ D)) \qquad or \tag{4}$$

$$(uop \ (A \ bop \ B)) \ bop \ (uop \ (C \ bop \ D)) \qquad or \tag{5}$$

$$(uop \ (uop \ (A))) \ bop \ (uop \ (B \ bop \ C)) \qquad or \tag{6}$$

$$(uop \ (uop \ (A))) \ bop \ (uop \ (uop \ (B))) \tag{7}$$

where A,B,C,D,E, and F $\in$ Set of matrices
bop $\in$ Set of binary matrix operators
uop $\in$ Set of unary matrix operators

would have the same task graph. In addition to this, a number of applications translates to the above task graph by the nature of application itself.

The experiments are conducted on a heterogeneous collection of unevenly loaded workstations. Hence, the time of completion of tasks are probabilistic. The average completion time registered by *T1* and *T2* on a representative single machine is between

20 and 25 mnts. The computational requirement of *T3* and *T4* are dependent upon the values of the input itself. This introduces one more probabilistic factor in the completion time of tasks. Consequently, *T3* and *T4* registers an average completion time in the range of 8 to 20 mnts under similar conditions. *T5* is executed when *T3* and *T4* completes its execution.

The task parallelism in the problem is by independent execution of two task parallel arms in the task graph. The data parallelism in the problem is by data parallel execution of each of the tasks *T1*, *T2*, *T3* and *T4*.

The first experiment is intended to show the effect of load fluctuations on the time of completion of tasks. Tasks *T1* to *T4* are executed data parallely on three nodes. The grain size assigned to a node is in accordinance with the load snapshot. The run is repeated 5 times. The summary of results is given in Table 1.

Table 1: **Some sample scenarios(time in mnts)**

No	T1	T2	T3	T4	CritPath	Diff_Crit_Path
1	7.9	9.1	5.3	4.7	13.8	-0.6
2	8.1	10.8	4.8	4.7	15.5	-2.6
3	11	8	7	5	18	+5.0
4	8	10.5	7.2	4.9	15.4	-0.2
5	11.2	8	4.8	7.1	16	+0.9

The time of completion of *T1* to *T4* is shown in Columns 2 to 5 of table 1. A task finishes its execution when all its subtasks finishes. Hence, the time of completion of a task is the time taken by its subtask which finishes last. The time of completion of an arm is the time taken to complete all the tasks in the arm. *CritPath* records the time taken by the arm which finishes last. *Diff_Crit_Path* records difference in the time of completion of two arms in the task graph.

The observations of interest from the experiments could be summarized as

- In spite of a task sub-division based on runtime load conditions, a considerable variation is registered in the time of completion of tasks. For instance, T1 registers a high of 11.2 mnts in the 5th observation against a low of 7.9 mnts in the first observation. The fluctuation amounts to 41% of the lower value.

- The values presented are without inducing any artificial loads. Under heavy load fluctuations, the values would fluctuate even more.

- The order of completion of task parallel arms is probabilistic. Also, the difference in their time of completion could be substantial. In the table, a negative value of *Diff_Crit_Path* signifies first arm finishing before second arm and vice versa.

$$Diff\_Crit\_Path = T_{arm1} - T_{arm2} \tag{8}$$

  where T_{arm1} and T_{arm2} are the times of completion of arm 1 and arm 2 respectively.

- The time of completion of individual tasks could have a cumulating or compensating effect on the difference in critical paths of the arms. Observation 3 shows the cumulating effect whereas observations 4 and 5 shows the compensating effect. The following discussion formalizes the observation.

  Consider N task parallel arms each consisting of M tasks. Let $t_{i,n,j}$ and $t_{i,l,j}$ be the times of completion of task j in arm i under no load and loaded conditions respectively. Under no load conditions, time of completion of a task is assumed to be constant (ie. $t_{i,n,j}$ = constant). Let $T_{i,n}$ and $T_{i,l}$ be the times of completion of task parallel arm i under no load and loaded conditions respectively.

$$T_{i,n} = \Sigma_{j=0}^{M} t_{i,n,j} = c \tag{9}$$

  where c is a constant.

$$T_{i,l} = \Sigma_{j=0}^{M} t_{i,l,j} \tag{10}$$

  Consider two arms a and b. The fluctuation in time of completion of tasks in an arm is said to compensate the fluctuation in the other if:

$$\Sigma_{i=0}^{M} Mod(t_{a,l,i} - t_{b,l,i}) > Mod(T_{a,l} - T_{b,l}) \tag{11}$$

  The net compensation in time *(tc)* is given by

$$t_c = \Sigma_{i=0}^{M} Mod(t_{a,l,i} - t_{b,l,i}) - Mod(T_{a,l} - T_{b,l}) \tag{12}$$

  If there is no compensating factor, the following relations will hold

$$\Sigma_{i=0}^{M} Mod(t_{a,l,i} - t_{b,l,i}) = Mod(T_{a,l} - T_{b,l}) \tag{13}$$

The effect of presupposing the order of completion of tasks to schedule subsequent tasks is shown in Table 2. Each row in the table is derived from the corresponding row in Table 1. The first column shows the time of completion of both arms if T1 is waited for before T2. The second column shows the time of completion of both arms if T2 is waited for before T1. The last column shows the time of completion with event driven scheduling.

Table 2: **Effect of various scheduling**

No	T3_T4	T4_T3	Event_driven
1	13.8	14.4	13.8
2	15.5	15.6	15.5
3	18	18	18
4	15.4	17.7	15.4
5	18.3	16	16

The observations of interest from the experiments could be summarized as

- When the difference in critical path compensates in an arm, presupposing the order of completion of tasks may affect the performance. In observation 5, the first presupposed order of completion fails. Similarly, in observation 4, the second presupposed order of completion fails.
- When the difference in critical path cumulates in an arm, presupposing the order of completion of tasks may not affect the performance. Observation 3 presents a case for the same.
- The event driven scheme always matches the best result among the results with presupposed orders of completion.

In the next experiment, the problem is executed by exploiting its data parallelism, task parallelism and both task and data parallelism. The data parallelism in the problem saturates when the granularity falls below a designated value. The task parallelism in the problem is limited by the number of task parallel arms.

The results are summarized in Table 3. The sequential time of execution is presented for comparison. The second column states the nature of parallelism exploited. NOP stands for *No Parallelism Exploited*, DP for *Data Parallelism Exploited*, TP for *Task Parallelism Exploited* and TDP for *Task and Data Parallelism Exploited.* Third column shows the number of machines utilized for parallel computation. Fourth column gives the time of completion of the problem. The last column gives the split up of the execution time.

Table 3: **Effect of Exploiting task and data parallelism**

No	Parallelism	#OfMachines	TimeOfCompletion (in mnts)	SplitUp
1	NOP	1	74.2	23 + 24 + 12 + 15.2
2	DP	2	41.7	14 + 14.2 + 6.5 + 7
3	DP	3	26.3	8 + 9 + 4.5 + 4.7
4	TP	2	36	24 + 12
5	TD	4	19.8	(13.3,12.7) + (6.5,5.2)
6	TD	6	13.2	(8.4,8.2,8.0) + (4.3,4.1,4.8)

The first row corresponds to the sequential execution of the problem. The split up shows the time taken for T1 to T4. The second and third row presents the results with data parallel execution on two and three nodes respectively. The split up shows the time taken for data parallel execution of T1 to T4. The fourth row shows the task parallel execution on two machines. The split up shows the time of tasks in the critical path (the arm which finishes last). The fifth and sixth rows show the results of exploiting both task and data parallelism. In the fifth row, each task is data parallelized into two nodes. In the sixth row, each task is data parallelized into three nodes. The split ups shows the time taken by data parallel subtasks of the tasks in the arm which proves to be the critical path.

The observations of interest from the experiments could be summarized as

- The problem is a case where task and data parallelism are complementary.
- The task parallelism in the problem saturates with the utilization of two nodes. This is because there are only two task parallel arms in the application.
- The data parallelism in the problem starts saturating with the utilization of three nodes. It could be seen from the third row that the granule size has reached around

four minutes of execution time. Further subdivisions does not improve performance because of the fixed time overheads in splitting the problem, migrating the code and arguments, compiling the code and collecting the results.

- It could be seen that by exploiting both task and data parallelism, six nodes are utilized for parallel execution before the same granule size of four minutes is reached. Hence, the optimum number of nodes utilized for parallel computation is increased.

7 Guidelines for composing user programs

The task graph of the application would be a directed graph with nodes representing tasks and edges representing precedence relationships. In the task graph, a task could be a *Starting task*, an *Intermediate task* or the *final task*. *Starting tasks* are those which do not have to meet any preconditions. *Intermediate tasks* and *Final task* have precedence constraints. The *Final task* takes care of program termination. Writing a program for the platform involves translation of the task graph to the final block structured code.

The *Final task* should take care of the termination of the program. Otherwise, the program would wait in an infinite loop for further events to come. If the application has more than one *Final task*, the methodology insists on introducing a pseudo final task after all the tasks. The pseudo final task should take care of termination.

Some tasks wait for the completion of more than one task. Such tasks can be placed inside the block of any of its predecessor task. In such cases, the choice is left to the programmer. However, it will not have any effect on the translated code. Readability of the code can be enhanced by placing appropriate comments whenever such discretions are made.

While composing programs with existing modules, each task could be available as separate program files. Flexibility in program structuring is permitted under such conditions. The only modification that is to be done in such cases is to wrap the code for each tasks with task demarcating constructs, *Task_begin* and *Task_end* along with their *OnFinish* directives.

The *Register_event* calls are inserted by the translator itself. The call has at least once semantics. A redundant insertion of the call by the programmer would be ignored.

While programming for anonymous execution, no assumption should be made about the underlying system. Though the portability of the system is provided, the portability of the migratable user program has to be ensured by the programmer himself.

When more than one parallel arm could be started on the occurance of an event, it would bring performance benefits to migrate the more time consuming arm to the least loaded node. Typically, in the integrated platform, each task is migrated to the then least loaded machine. However, the translator inserts the *ARC_function_call*s for the tasks in the order specified by the programmer. Hence the programmer should take care in placing the more time consuming task in front of others.

8 Future work

Further challenge in transparent platforms for NOWs is to support communicating parallel tasks. The key issue in such an attempt is to permit intertask communication in the premises of distribution transparency. Such an attempt can address problems with patterns in their process interaction. Optimizations possible with different interaction schemes could be explored.

9 Acknowledgements

We thank Dr. Hema A. Murthy, Asst. Prof., Dept. of CSE, IIT Madras and Shajit Iqbal, Research Scholar, Dept. of CSE, IIT Madras for their inputs related to speech signal processing.

References

[1] P.J.Hatcher and M.J.Quinn, Data-Parallel Programming on MIMD Computers, *The MIT Press*, Cambridge, MA, 1991.

[2] Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis, Introduction to Parallel Computing, *The Benjamin/Cummings Publishing Company Inc.*, 1994.

[3] Bradley K. Seevers, Micheal J. Quinn, Philip J. Hatcher, "A Parallel Programming Environment Supporting Multiple Data Parallel Modules", *SIGPLAN*, pp 44-47, Jan 1993.

[4] Thomas Gross, David R. O'Hallaron, and Jaspal Subhlok, "Task Parallelism in a High Performance Fortran Framework", *IEEE Parallel and Distributed Technology*, Vol.2, No.3, pp 16-26, Fall 1994.

[5] G.McRae, A.Russell, and R.Harley, "CIT Photochemical Airshed Model: Systems Manual", 1992.

[6] Barbara Chapman, Hans Zima, Piyush Mehrotra, "Extending HPF for Advanced Data Parallel Applications", *IEEE Parallel and Distributed Technology*, Vol.2, No.3, pp 59-70, Fall 1994.

[7] Ian Foster, "Task Parallelism and High-Performance Languages", *IEEE Parallel and Distributed Technology*, Vol.2, No.3, pp 27-36, Fall 1994.

[8] Henri E. Bal, Mathew Haines, "Approaches for Integrating Task and Data Parallelism", *IEEE Concurrency*, pp. 74-84, Jul-Sep 1998.

[9] Rushikesh K. Joshi, D. Janaki Ram, "Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations", To appear in *IEEE Trans. on Software Engineering*, Vol.25, No.1, Jan/Feb 1999.

[10] R. Parthasarathy, "Designing a Robust Runtime System for ARC", Project report, Acc.No.97-BT-04, Dept. of Computer Science and Engineering, IIT Madras, 1997.

[11] R.P. Ramachandran, M.S. Zilovic, R.J. Mammone, "A comparitive study of Robust LP Analysis Methods with Applications to Speaker Identification", *IEEE Trans. Speech, Audio Processing*, Vol.3, pp. 117-125, Mar 1995.

[12] D.A.Reynolds, R.C.Rose, "Robust Text-Independent Speaker Identification using Gaussian Mixture Speaker Models", *IEEE Trans. Speech, Audio Processing*, Vol.3, pp.72-83, Jan 1995.

[13] C.Chandrasekhar, B.Yegnanarayana and R.Sundar, "A constraint satisfaction model for recognition of Stop Consonant-Vowel(SCV) utterances in Indian languages, *Proc. Int. Conf. on Communication Technologies(CT-96)*, Indian Institute of Science, Bangalore, pp 134-139, Dec 1996.