Integrating Task Parallelism in Data Parallel Languages for Parallel Programming on 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 designing applications for non-determinism in the order of completion of constituting tasks. Design of the runtime support as well as system level bookkeeping is discussed. The model is general enough to be applied 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 is presented. The performance related aspects are also discussed.

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. The control 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 control parallelism[3]. This kind of 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 and may be implemented for parallel computation[4]. As an example, the airshed model is a Grand Challenge Application that characterizes the formation of air pollution as the interaction between wind and reactions among various chemical species[5].

- *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 control parallelism in them.

- *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 discovered by a compiler[7].

- *Problem characteristics*: Many problems can benefit from a mixed approach, for instance, with a task parallel coordination layer integrating multiple data parallel computations. Some problems admit both data and task parallel solutions, with the better solution depending on machine characteristics or personal taste[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 control parallelism. Hence, it is desirable for a parallel program to exploit both data and task parallelism inherent in a problem. A parallel programming environment should provide adequate support for both data and task parallelism to make this possible[8].

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.

- *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 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. In the model, the system takes the responsibility to intimate the user process when an event of its interest occurs. Events of interests signifies completion of one or more tasks which meets the preconditions for another task that is waiting to be executed. This takes care of the probability factor in the order of completion of tasks that constitute the program. The model provides for templates by which events of interest could be registered with the system.

The model aims at a parallel programming platform that permits expressibility for 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. This poses some challenges. The data parallel model of computation of these platforms could be different. Consequently, the program structure favored by these platforms would be different. The system services provided by them viz. migration, result collection etc could also be different. Hence, at the programming level, the model restricts itself to expressing tasks and their precedence relationships. This makes sure that an integration of the model to an existing data parallel platform does not contradict its existing goals and the rules by which it handles data parallel computation. Also, it inflicts only minimal disturbance to the existing system.

Another issue which crops up during the integration pertains to data parallel subdivision of divisible tasks. The underlying data parallel model could be subdividing a data parallel task into subtasks. These subtasks could be migrated to different nodes over the network. The underlying data parallel platform could be considering subtasks as separate entities from the time it is spawned to its result collection. The platform would have had no purpose to identify a task as a collection of subtasks that constitute it. But, the completion of a task could be a significant event in the proposed integrated model since the system has to intimate the user process when an event of its interest occurs. The completion of a task depends upon the completion of the subtasks into which it is split by the underlying data parallel model. Hence, it becomes indispensable to integrate to the existing system, the notion of a task as a collection of subtasks to which it is divided.

The program expressibility of the model reflects the task parallel blocks and precedence relationships in the task graph. Two constructs viz. *Task_begin* and *Task_and* are introduced to demarcate the blocks in the block structured code. Another construct, viz. *OnFinish* is provided to specify the preconditions of tasks. The syntax and semantics of these constructs are described below.

```
Task\_begin(char *TaskName) OnFinish(char *WaitforTask, ...)

This marks the beginning of a block. Each block signifies a task. The
task name of the task it signifies is the only argument to the construct.
If the task has precedence relationships to be met, the Task_begin has
to be immediately followed by another construct viz. {\em OnFinish} with the
names of tasks it has to wait for as its arguments. If {\em OnFinish} is not
furnished, the parser will presume that there are no preconditions to the
task. {\em OnFinish} could have variable length of arguments.

Task\_end(char *TaskName)

This marks the end of a block. The only argument is the name of the
task the block signifies.
```

2.3 Program Structure and Translation of a task graph

A sample task graph and its block structured code is shown to illustrate the expressibility provided by the model. Also, it illustrates translation of a given task graph into the program structure favored by the model. The task graph given in figure 1 is used for the same.

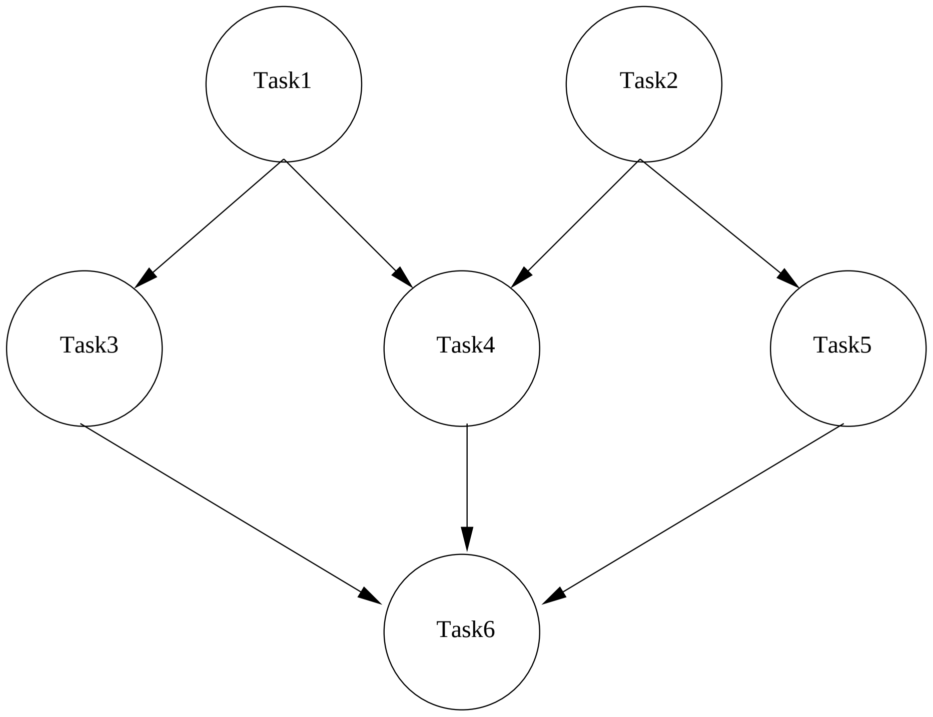

Figure 1: **A sample task graph with precedence relationships**

```
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)
```

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

The outline program given above shows the expressibility of a task graph in the model. At the first level, *task1* and *task2* could be executed in a control parallel fashion at the beginning of the run itself. This is evident from their *Task_begin* constructs, since it is not followed by the construct *OnFinish*. A task which has to wait for another task to finish is written inside its predecessor task's block. Also, the *Task_begin* construct of such tasks would be immediately followed by the construct *OnFinish* which specifies the preconditions. In case where a task has to wait for more than one task, it can be placed inside one of those blocks which represents a predecessor task. But, its *OnFinish* should carry the name of all its immediate predecessors This could be seen from the *OnFinish* directives of task4 and task6.

Typically data parallel platforms split divisible tasks to subtasks. Hence the model provides for a call viz. *task()* to register a task as a collection of subtasks. The syntax and semantics of the call is described in section 3.3.

2.4 Separation of System's and Programmer's concern

The system is organized such that the user process registers events of interest (completion of one or more tasks) which are required in order to meet the precedence constraints and the system in turn signals the user process on occurrence of any registered event. This relieves the programmer from designing applications for non determinism in the order of completion of tasks.

With the program translation capabilities that is provided, the model relieves the programmer from network related code. Another responsibility rested upon the translator is to contend with the conflicting interests of expressibility and performance. Arguments from the expressibility point of view favor a control structure that best reflects the precedence relationships in the code. This in turn would 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 structure suitably describes the task graph and contributes to the elegance of the code. But, it does not support non-determinism in the order of completion of tasks according to the flow of control permitted by traditional languages. 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 that well answers the concerns of non determinism.

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. The parser and sample translated code is described in section 4.1.

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 control 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 program to get the current availability and load of machines. The user program can use this information, along with the machine handles returned by the call to migrate his RIBs to least loaded machines. For data parallel modules, the load ratio on machines could be used to decompose the domain of computation to achieve load balancing. Subsequently, each grain of computation can be migrated to the corresponding machines, along with the relevant initial data. The ARC model provides two calls for functionalities related to result collection. Availing these services, the user program can get the status of a submitted task as well as collect the results when they are ready. 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 various machines available in the system 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 is 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 using 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. Return value is 1 if the results are available,
else it is -1. When the results are available, it can be collected by
obtain_results call.
```

3.2 Outline of ARC runtime support

This subsection outlines the runtime support of ARC[10]. The support consists of a daemon running on each machine present in the pool of machines which cooperate for parallel computation. This daemon is termed *local-coordinator* (lc). It is the responsibility of the *lc* to coordinate the processes initiated on the machine on which it runs. Any process which intends to make use of the system services registers itself with the *lc* that runs on its machine. This is done by the call *initialize_ARC*. Complementing the call is *close_ARC* which de-registers the user process from the system. 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 program
with the runtime system. In response to this call, the 'lc' allots an
exclusive communication channel between the 'lc' and the user program for
subsequent communication.

void close_ARC(void);

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

One of the machines in the pool is selected to run a daemon which coordinates the local-coordinators of all the machines which participate in the pool. This daemon is termed system coordinator *sc*. The *sc* keeps track of the *lcs* in the pool and hence the machines participating in the pool. It also facilitates communication between individual local-coordinators. The local-coordinator and system coordinator communicate by virtue of predefined messages (through a dedicated channel). Similarly, the user process communicates with the local coordinator by predefined messages. In a typical session, a message session sequence would be initiated by the user process by sending a message to its *lc* . The *lc*, in turn, sends the appropriate message to the *sc*. The *sc* may either reply to this message or send a message to another *lc* if required. In the first case, the *lc* on receiving the message would generate and send a message to the use process which initiated the sequence. In the latter case, the *lc* which receives the message from the *sc* would send a reply on completion of the responsibility. This message would be passed to the *lc* which initiated the sequence. The local coordinator would generate and send a message to the actual user process which initiated the message sequence.

3.3 The Integrated Platform

The integration of the model into the ARC framework includes providing for additional function calls and modifications to the the existing runtime support. The crux of such modifications are presented in the section.

The additional calls supported include a call to register a task as a collection of subtasks, calls to initialize and close the system and a call to register events of interest with the system. The ARC model permits the decision of the number of subtasks at run time. Also different tasks could be divided into different number of grains. For these reasons, the call to register a task as a collection of its subtasks, viz. *task()*, provides for a variable length argument. Responsibilities of the calls to initialize and close the system viz. *TaskInit()* and *TaskClose()* are trivial. The call to register event viz. *RegisterEvent()* is inserted by the parser 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 subtasks. The first
argument is the task name itself. The second argument is an integer which
denotes a subtask. If the task is not subdivided, the task_no itself is
to be furnished here. If the task is subdivided, there could be more
arguments each of which represents 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 identifier. Following it is a variable length argument list of task
names constituting the event. The return value is either SUCCESS of ERRNO
corresponding to the error.
```

The modifications to the runtime support of ARC could be summarized as additional table management, additional message protocols between daemons and minimal changes to system's response to some existing messages. Additional table management includes mapping tasks to their subtasks, storing the events of interest for a process and status of finished tasks. Additional message protocols are those to access and modify these tables. There has been an inevitable change to existing system. This was to make the system generate a message to a user process when an event of its interest occurs. In ARC, a task can be considered to have finished its execution only when all its subtasks finish their execution. The local *lc* gets a message from remote *lc*s when the subtasks given to them finish their execution. The ARC *lc* stores the result and waits for the user process to ask for it. In the integrated system, the *lc* has an additional responsibility when a subtask returns. It checks if the subtask that has finished is the last subtask of the task. If the task can be considered to have finished its execution, it would check the status of finished tasks and events of interest to find out if any event of interest to the user process has occurred with the completion of the task. If an event of interest has occurred with the completion of the task, it initiates a message sequence to intimate the event to the corresponding user process. Additional messages are defined in order to achieve the same.

3.4 A sample block in the integrated platform

The program structure of the integrated platform would be same as the one shown with the model. It has been mentioned earlier that the code inside a block would be meant for the platform to which the model is integrated. A sample block is given to provide a comprehensive view of the integrated platform. The semantics of the calls have been discussed in section 3.1.

```
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)

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

The code shows how ARC decides the number of sizes of grains of computation at runtime after collecting the load information. Also, note that the arguments to the *task()* call reflect the actual division employed. The block is marked by *Task_begin* and *Task_end*. *OnFinish* specifies the precondition of TaskN as the completion of TaskM.

4 Design and Implementation

The parser for program translation, the local coordinator daemon for user process coordination, the system coordinator daemon for coordination of the pool of workstations and functional library support to avail system services are the constituent elements of the system.

4.1 Parser

The parser translates the program submitted by user to the final runnable program. This involves insertion of appropriate network related code, translation of the pseudo control structure provided by the model to one which is supported by the native language, construction of events of interest to the user process and transparent insertion of some system service calls.

In the first scan of the user submitted program, parser constructs the precedence graph. It marks the tasks which do not have any precedence relationship to be met. It finds the events of interest to the user process. The calls to register these events with the system are inserted in the code. A header file is generated to define the event names and the file is included in the user program file. This permits event names itself to be used as cases in the final flat switch-case structure. The tasks which do not have to meet any precedence relationships are extracted and placed before the flat switch-case structure since they could be executed without waiting for any events.

In the second scan, the parser constructs an infinite loop which waits on a socket to read event interrupts. Inside the loop is placed a switch case structure with event names as cases. The body of a case is the code fragment of the task which waits for the event. This control structure is a flat one unlike the control structure permitted by the model. The termination would become part of the case which corresponds to the final task.

The prototypic version of parser developed can convert the code only to one native language, viz. C. It is relatively straightforward to extend the scope of the parser to cater to other native languages.

Below given is a sample translated code. The code is obtained by translation of Pseudo Code 1. The transformations by the parser could be seen by mapping the sample translated code with 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;

     }
}

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

4.2 Local Co-ordinator daemon

Each workstation which enrolls in the pool of machines for parallel computation runs a daemon viz. localdaemon(lc). The user processes communicate and interact with the system through the *lc*. Servicing requests from the user processes, linking the node on which it is run to the system and book keeping for the user process running on its machine are the responsibilities of *lc*.

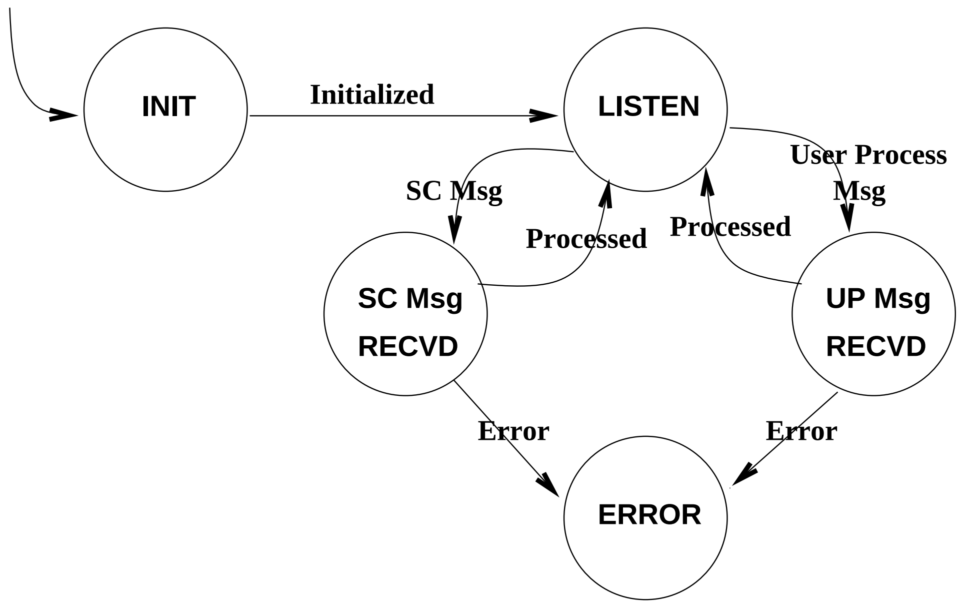

Figure 2: **FSM of LC**

Fig. 1 gives the finite state machine of the *lc*. In the **INIT** state, *lc* initializes its data structures and cleans up the auxiliary system files left behind by and earlier *lc* process. After this, it establishes a TCP socket connection with the *sc* and registers itself with the *sc*. In **LISTEN** state, the *lc* checks for messages from the *sc* and waits for user programs to register with the system. On a message from *sc*, it transits to **SC Msg RECVD** state where it services the message. On a message from a user process, it transits to **UP Msg RECVD** where it services the request from user process.

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. Once it is registered with the local coordinator, it is given an exclusive communication channel though which subsequent communication is effected. When the local coordinator is initiated, the communication with the system coordinator is effected by giving the address of the machine on which the system coordinator is run as a command line argument.

The tables maintained by the *lc* could be distinguished as those required to support task parallelism and others. This separation eases a clean integration of task parallelism into the existing systems. In ARC, the local coordinator maintained three tables, viz viz. *Program and Task Table(PTT)*, *Recovery Information Table(RIT)*, and *Results List Table(RLT)*. PTT maintains the mapping between the process ids of the processes on the machine and the socket descriptors of the sockets that connects the processes to the *lc*. It also demarcates processes as user processes initiated on the node and tasks submitted by user processes on other nodes. RIT maintains information that is relevant for recovery in the event of a failure. RLT stores the results of the tasks submitted by user processes on the machine as and when they are available. This is stored till it is claimed by the user process.

It could be seen that the system does not keep track of the subtasks which constitute a task. The significance of keeping track of subtasks that constitute a task was already discussed. Hence, a new table, viz. *Task Table (TT)* is maintained which maps tasks to the subtasks into which it is divided. This mapping is done on a per process basis. The table is updated when a task is subdivided. The programming language interface for updation of the table has already been discussed.

In the integrated system, the *lc* keeps track of the events of interest to the user processes. It maintains a table, viz. *Event Table (ET)* for the purpose. It stores the predeclared events of interest on a per process basis. The interface for updation of the table is inserted by the parser by interpreting the events of interest from the user code.

Other than the additions of the above mentioned tables, their programming language level interfaces, and the message sequences that it initiates, there are some modifications to the existing *lc*. This is to make the *lc* inform the user process when an event of interest occurs. A task finishes its execution when the last subtask of the task finishes. Hence, when the *lc* gets an intimation of completion of a subtask from the *sc*, it checks if the task has finished its execution with the subtask that has returned. If the task is over, the *lc* checks to see if it triggers any event of interest. If so, the *lc* initiates a message to the user process intimating the occurance of the event. *TT* and *ET* are the tables consulted by the *lc* in order to accomplish the job. In cases where completion of a subtask results in more than one event of interest, they are intimated with separate messages one after the other. Below given are the structure of Event Table and Task Table.

```
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 indentifiers
BOOLEAN * SubTaskOver;           // Sub task completion status
int NumberofSubTask;             // Number of subtasks

}
```

4.3 System Co-ordinator daemon

The System Co-ordinator(sc) coordinates the set of *lc*s that constitutes the system. The *sc* maintains the global information of the system. Also, for small sessions, the *sc* routes the messages between *lc*s so that the overhead for frequent connection establishment and closing is minimized. The *sc* is connected to *lc*s through TCP sockets. The message structure includes a field to indicate the destination address in order to facilitate this. The *sc* that is employed by the ARC is a fairly thin deamon with minimal information stored and hence scales up quite well with respect to the number of *lc*s that participate in the pool.

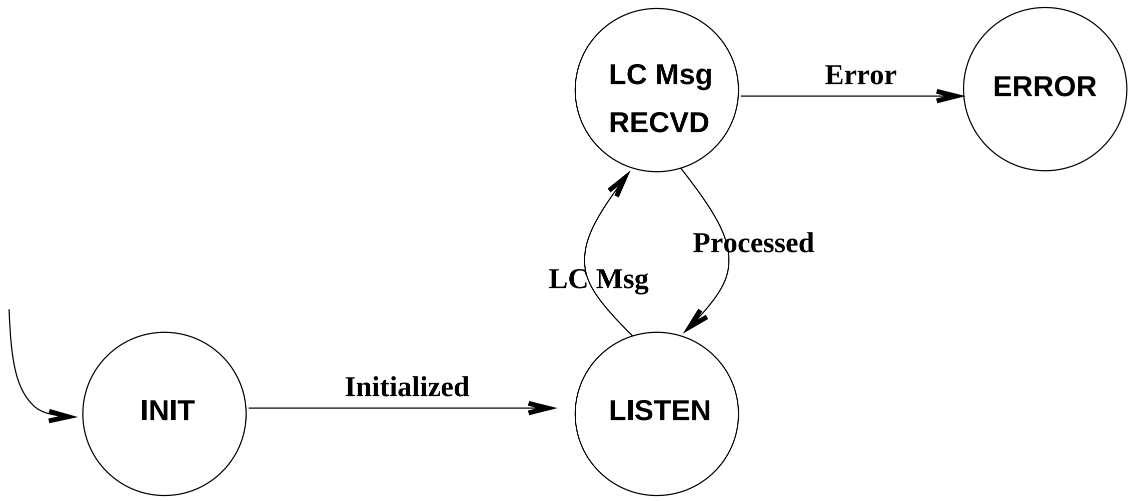

Figure 3: **FSM of SC**

In the state **INIT**, *sc* cleans up the current directory for any auxiliary system files left behind by an earlier *sc* process. It then initializes its structures. In the **LISTEN** state, the *sc* polls for connection requests from *lcs* and registers them with the system, establishing a TCP socket between itself and the *lc*. It then listens for messages from the *lcs* on these exclusive channels, and services the requests. The *sc* remains in this state throughout its lifetime or till it encounters an error.

Our implementation for the integration of task parallelism into ARC does not disturb the existing *sc*.

5 Applications

Applications with coarse grain control parallelism or coarse grain data parallelism or both are the target of our platform. The motivation of the work has already discussed some broad classes of applications that could potentially benefit from the platform. In addition, here we will discuss some specific applications.

Many applications in signal processing have potential coarse grain control and data parallelism. A good proportion of signal processing applications does some form of signal transformation. These transformations are typically followed by a filtering module which filters the transformed signal. The inverse of the transformation could follow the filtering stage. As an example, an application could consist of a Fourier transform followed by some filter module that is dependent on the purpose of the application which is followed by the inverse Fourier transform. Fourier transform and its inverse are amenable to data parallel computation. So is the case with many other signal transforms. The exploitation of parallelism in signal processing becomes even more important in real time signal processing.

Here, we discuss exploitation of parallelism in one such applications, viz. Speaker Verification Problem. The speaker verification program starts with a sample of the time domain signal. A solution to the problem starts with a linear predictive analysis[11] of the input time domain signal to yields linear predictive(LP) co-efficients. From the set of LP co-efficients is obtained LP Cepstrums which are features of the input signal. There are different methods to obtain evidences for verification from the LP Cepstrums. Gaussian Mixture Model(GMM) method[12], Neural Network Methods etc. are some examples. Some methods prove better than the rest according to the nature of input set. Different methods could be applied control parallely on the same set of LP Cepstrums. *Constraint Satisfaction Model*[13] combines evidences obtained from each of these models. Each of the methods, in turn, could exploit the data parallelism in it.

Below given is a sample pseudo-code for the application on the integrated platform.

```
LPAnalyse();
ComputeLPCepstrums();

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

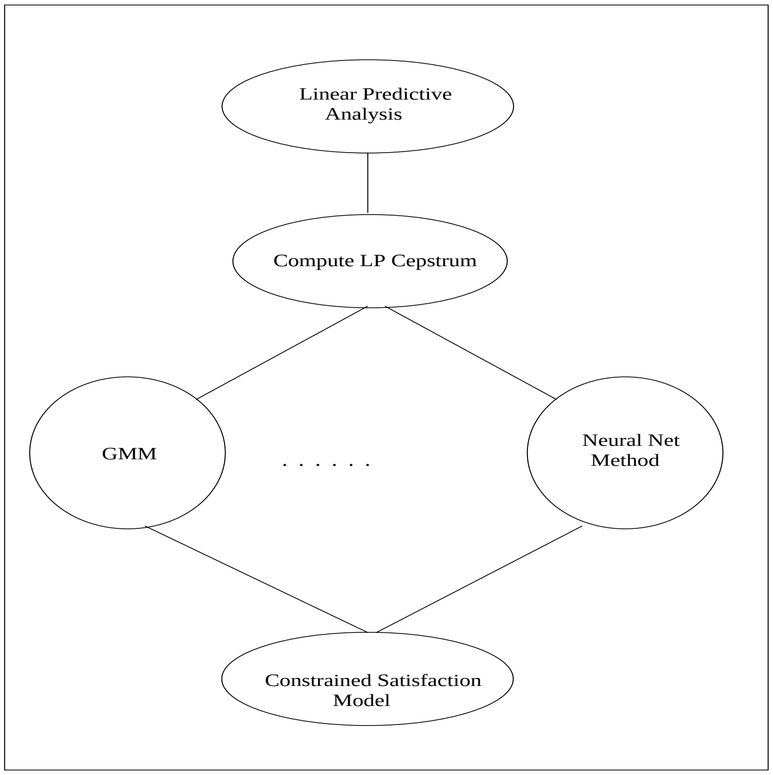

Figure 4: **Task graph of the application**

```
// 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)
```

The sample application that is chosen has a relatively simple precedence graph. One reason for the same was to keep the pseudo-code as compact as possible.

6 Performance Analysis

The section presents performance related aspects of the work. The test bed for the experiments consists of a heterogeneous collection of interconnected workstations with other load coexisting. The problem that is considered has control as well as data parallelism. The task graph of the problem is given in figure 5.

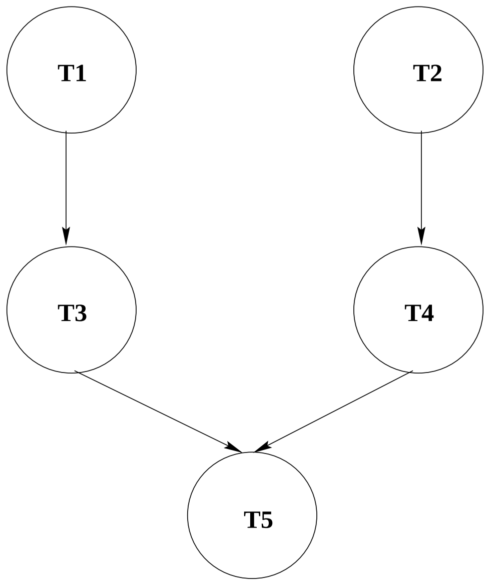

Figure 5: **Task graph of the application**

As the program starts its run, the two tasks viz. T1 and T2 can start its execution. These two tasks do matrix multiplication on two different sets of large matrices. Task T3 is a user defined function to operate on the resultant matrix of T1. Similarly T4 is a user defined function to operate on the resultant matrix of T2. T3 and T4 can start execution only after the respective completion of T1 and T2. The average completion time registered by T1 and T2 on a representative single machine falls between 20 and 25 mnts. These measurement comes at relatively lightly loaded CPU conditions though there could be spurts of loads that occur during the period of run. The computational requirement of T3 and T4 are dependent upon the values of the input itself. This would cause another probabilistic factor in the completion time of these tasks. Consequently, T3 and T4 registers an average completion time in the range of 8 to 20 mnts under same conditions. T5 is executed when T3 and T4 completes its execution. In the problem considered, T5 is a thin task with the only responsibility being collecting the results of its two predecessor tasks. The task completion time of T5 is insignificant and hence not considered for the analysis.

The control parallelism in the problem is by independent execution of two control 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.

In the first experiment, tasks T1 to T4 are data parallely executed on three nodes each. The run is repeated 5 times and the time of completion of each of the tasks is observed. During data parallel division, the then load conditions are taken. The grain size of individual data parallel subtasks is determined using this load snap shots.

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

No	Crit_T1	Crit_T2	Crit_T3	Crit_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 task is shown in the table as its critical path. The terminology is adopted because a task is said to have completed when the last among its subtasks completes. Along with the time of completion of T1 to T4 is given the larger value of time of completion of the first and second arm of the task graph. The last column gives the difference in the time of completion of the two arms.

The observations of interest from the experiments could be summarized as

- In spite of a task division policy based on runtime load conditions, the completion time of tasks could vary considerably. It can be seen from the table that T1 registers a high of 11.2 in the 5th observation against a low of 7.9 in the first observation, yielding a difference of 3.3 mnts, which is 41% of the lower value.
- Any of the control parallel arms could finish before the other and the difference in their completion time could be substantial. A negative value of difference in critical path signifies the first arm finishing before the second and vice versa.
- The difference in critical paths of the arms could have cumulating or compensating effects from the individual completion times of the tasks in the arms. Observation 3 shows the cumulating effect whereas observations 4 and 5 shows the compensating effect.
- The values presented are taken without inducing any artificial loads. Under heavy load fluctuations or with artificially altering load, the probabilistic values would fluctuate even more.

The earlier experiment has brought out the probabilistic factors in the time of completion of tasks. The next set of values would present the effect of presupposing the order of completion of tasks to schedule the subsequent tasks. In the table given, each row is derived from the corresponding row of the last table. The first column shows the critical path if T1 is waited for before T2 and the second column if the expected sequence is opposite. The last column shows an event driven model.

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, the difference in the two scheduling schemes would be more. Observations 4 and 5 presents cases for the same.
- When the difference in critical path cumulates in an arm, there will not be any difference in the two scheduling schemes since the critical arm remains critical however it is scheduled. Observation 3 presents a case for the same.
- The event driven scheduling always gives the same performance as the better of the two schemes. As the task graph gets more complicated, there would be more possible schemes and only one of them would perform as good as the event driven model.
- The scheduling schemes that are referred above are mainly targeted at scheduling the various arms of the task graph in an integrated task-data parallel model. These strategies do not stem from a perspective of scheduling for load balancing.

The next experiment is conducted to show the effect of exploiting data parallelism, task parallelism and both task and data parallelism. For comparison, the sequential time of execution is also presented. 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.* The other columns show the number of machines utilized for parallel computation and the time of completion of the problem. The split up of the time of completion is also shown.

Table 3: **Effect of Exploiting task and data parallelism(time in mnts)**

No	Parallelism	#OfMachines	TimeOfCompletion	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 of total time is the time taken for T1 to T4 respectively. The second and third row presents the results with data parallel execution on two and three nodes respectively. The split up in this case also is the time taken for T1 to T4 respectively. The scaling down of the time of execution is accounted by the data parallel execution of tasks. The fourth row shows the task parallel execution on two machines. The split up in this case is the time of tasks in the critical path. The last two runs employs both task and data parallelism with two nodes per task and three nodes per task respectively. The split ups in these cases are 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 the task and data parallelism are complementary.
- The control parallelism in the problem saturates with the utilization of two nodes. This is because there are only two control 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 the granule size has reached

around four minutes of execution time. Further subdivisions for parallelism does not yield results because of the fixed time overheads of splitting the problem, migrating the code and arguments, compiling the code and collecting the results.

- It could be seen that with the exploitation of both task and data parallelism, six nodes are utilized for parallel execution before the same granule size of four minutes is reached.

7 Guidelines for composing user programs

Composing a program over the platform involves translation of task graph of the application to the final code. A task graph can be expressed as a directed graph with nodes representing tasks and links precedence relationships. 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 for their execution. *Intermediate tasks* have precedence constraints to honor. The *Final task* is one which is responsible for termination of the program.

The *Task_begin* of *Starting tasks* need no *OnFinish* directives whereas it is required for the *Intermediate tasks* and the *Final task*. The *Final task* should take care of the termination of the program. Else, the program will wait in an infinite loop for further events to come. In cases where there are more than one *final task*, the methodology insist on keeping a single pseudo final task which takes care of termination.

It was mentioned earlier that the expressibility provided employs block structuring of the code. A block designates a task and could contain other blocks in it. Blocks written inside a block are those which can execute only after the completion of outer block. The complete precedence requirements including the implications from the structure of the code has to be specified.

Tasks which has more than one precedence relationship to be met could be written as a block inside any one of its parent blocks. In such cases, the choice of the parent block is left to programmer's discretion. However, the placing of such blocks would not have any effect on the translated code. It should be noted that such tasks should not be placed in more than one parent blocks. Also, readability of the code can be enhanced by placing appropriate comments wherever such discretions are made.

While composing programs with existing modules, programs for each tasks could be available as individual files. In such cases, the strucuring policy need not be strictly followed. 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.

Passing of arguments to tasks should be done keeping in mind the syntax, semantics and limitations of the call which supports it. Some systems are not designed for the tasks to take more than one stream as argument. In such cases, the programmer has to explicitly pack the argument streams before passing the argument and unpack it in the target task.

The *Register_event* calls are inserted by the parser itself. The call has atleast 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.

8 Related Work

There have been a few attempts at integrating task and data parallelism in the literature. Notable ones include Opus [14], Fx [15], Data Parallel Orca [16] and Braid [17]. Opus integrates task parallel constructs into data parallel High Performance Fortran (HPF). A task in Opus is defined as a data parallel program in execution. Due to this heavyweight notion of a task, inter-task communication is costly and hence Opus is suited only for coarse grained parallelism. Fx also adds task parallel constructs into HPF. But it uses directives to support task parallelism. It does not allow arbitrary nesting of task and data parallelism. Data Parallel Orca uses language constructs to integrate data parallelism into task parallel Orca. It has a limited notion of data parallelism as it does not support operations that use multiple arrays. This is because Orca applies operations only to single objects. Braid is a data parallel extension to task parallel Mentat. It uses annotations to determine which data an operation needs. The four models are cases of integrating task and data parallelism in specific languages. The proposed model, however is a generalized methodology to integrate task parallelism into *any* data parallel language. The model targets coarse grain task-data parallel computing on loaded NOWs. Further it supports arbitrary nesting of task and data parallelism unlike the existing models. The table 4 summarizes the comparison of our model with the existing works.

Table 4: Comparison of various task-data parallel integration models

System	**Fx**	**Opus**	**Data Parallel Orca**	**Braid**	**Our Model**
Aim	Integrate task parallelism into data parallel HPF	Integrate task parallelism into HPF	Integrate data parallelism into task parallel orca	Integrate data parallelism into task parallel mentat	Integrate task parallelism into *any* data parallel language
Basis of Implementation	Compiler	Run Time System(RTS)	RTS	Annotation	RTS
Expressibility (exp)	Restricted form of task parallelism	Full exp	Restricted form of data parallelism	Full exp	Full exp
Communication between tasks	Shared address space	Shared object	Shared object	Shared object	Dependent on native data parallel language
Grain Size	Fine	Coarse	Fine	Fine	Coarse

9 Future work

Further challenge in transparent platforms for NOWs is to support communicating parallel tasks. The key issue in such an attempt is to provide message passing abstractions 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.

A generic specification scheme for programs to specify the constraints on task/subtask mapping could be explored. This, along with anonymous execution will make it possible for the programmer to exploit the best of both. The constraints could be inflexible when a piece of code makes assumptions on the underlying system. The set of constraints could also include directives by the programmer of some hidden possibilities of optimizations that could be exploited. An approach to characterize nodes in a pool of machines keeping in mind the spectrum of distributed computations could be attempted.

Acknowledgements

We thank vijay and all other members of the Distributed & Object Systems Lab at IIT Madras for their support.

Appendix

The section describes the messages that are exchanged between the daemons and the user processes. Message exchanges occur between user process and the local coordinator on its machine, local coordinators and system coordinator and between a local coordinator and the processes which has migrated to its machine for execution.

The messages received by the *lc* are:

From the user program:

- get load factor: Prior to the submission of a set of tasks, a user program requests for the load information of a required number of nodes. In response to this, the *lc* forwards this message to the *sc*.

- take work: This is a task submission message. The destination is indicated in the message itself, and this is followed by the string representing the RIB's source file name and the arguments to the task. The *lc* forwards this set of messages to the *sc* after recording the information required for recovery.

- check results: This message is sent when the user program calls the *peek_results()* function. The *lc* checks in its structures for the presence of the required results and sends a corresponding reply to the user program. The content of this reply is the return value of the function.

- want results: This message is sent when the user program calls the *obtain_results()* function call. In response to this, the *lc* checks in its structures for the presence of the required results and sends a corresponding reply to the user program. The function call is blocking and sends this message repeatedly after a waiting period, incorporating task re-submissions if necessary.

- user is terminating: This message says that the user program is terminating. In response to this, the *lc* invalidates the user program related information maintained in its structures.

- register event: This message is used by the user program to inform *lc*, the events of its interest. In response to this, the *lc* updates the event table corresponding to the user process.

- register tasks: This message is used by the user program to declare a task as a collection of subtasks. In response to this, the *lc* updates the task table corresponding to the user process.

From the *sc*:

- take load factor: This message is in response to an earlier *get load factor* message sent on behalf of a user-program. It is followed by a message which represents the actual load factors. This set of messages is forwarded to the user program.
- take work: This message represents a task submission. It is followed by the source file name for the task and the arguments to the task. The *lc* checks whether the said task is already running, ie. whether the current message represents a resubmission. If not, or it the task failed, the *lc* compiles the source file and spawns a task process to execute the task.
- take results: This message represents the results of an earlier submitted task. It is followed by the actual result. The *lc* stores the result in its structures, so that it can immediately respond to a *want results* or *check results* from the user program. Also, with each of the take results, the *lc* checks if some event of interest registered in the event table has occurred. If so, it informs the user process by generating an *event occurred* message.
- generator failed: This message indicates that a particular user program has failed. The *lc* uses it recovery information to track all the tasks created by this particular user program and kills them, since they are no longer useful.
- LC is terminating: This message indicates that a particular *lc* has failed. The *lc* uses its recovery information to track all the tasks created by the user programs on the machine and kills them.
- terminate: This message is sent by the *sc* asking *lc* to quit. In response to this, the *lc* quits.

From the task process:

- take results: This is a message from the task process saying that it has completed the task assigned to it. This message is followed by the actual results of task execution, after which the task process exists. The *lc* forwards the result to the *sc*, after which, it invalidates the recovery information maintained for the task.

- give arguments: This message is sent by the task process as soon as it comes up. It is a request for the arguments of the task. The *lc* responds with a *take arguments* messages followed by the actual arguments for the task.

The messages received by *sc* are:

- get load factor: This messages is sent by an *lc* to the *sc* prior to the submission of tasks to the system. The purpose of this message is to ask for a certain number of machines in order to perform tasks. In response to this, the *sc* finds that many number of least loaded machines from the *sc* structures, orders their load factors in ascending order, and sends it to the requesting *lc*.

- take load factor: This message is sent by an *lc* to the *sc* to inform it of the current load factor on its machine. In response to this, the *sc* makes a record of this information in its structures.

- take work: This message, initiated by a user program is forwarded by an *lc* to the *sc*. It represents a task submission. It is followed by the string representing the RIB source file name and the arguments to the task. In response to this, the *sc* sends this set of messages to the appropriate *lc* as indicated in the first message.

- get code: This message is sent from an *lc* to the *sc* and is meant for another *lc*. It represents a request for the source code for a migratable module, and will be needed if the two machines in question are not on Network File System. In response to this, the *sc* forwards this to the corresponding *lc*.

- take code: This message is the counterpart to the previous message. The *sc* handles this message similar to the previous one.

- take results: This message is sent by an *lc* to the *sc* and represents the results of a task submission. This is followed by a message containing raw data, which represents the actual results. In response to this, the *sc* forwards this set of messages to the appropriate *lc*.

- generator failed: This message is sent by an *lc* to the *sc* informing it that a particular use program on its machine has failed. This is helpful because the tasks submitted by the user program are no longer needed to be executed, and hence can be killed, removing a possibly substantial workload. In response to this, the message is forwarded to the corresponding *lcs*.

- LC is terminating: This fact is recorded by the *sc* either when an *lc* terminated on its own accord or failed suddenly, or the network connection to it failed. In response to this, the *sc* broadcasts this message to all the other *lcs*

All the messages that a user program receives are from *lc*. The messages received by the user program are:

- take load factor: This is in response to one of the earlier *get load factor* message generated by the user program. It gives the required machine indices and their load factors to the user program.
- result availability: This is in response to a *check results* message generated by the user program. This messages lets the the user program know if the results are available or not.
- take results: This is in response to a *want results* message generated by the user program. This message passes the results to the user program.
- event occurred: This message is used to let the user program know the occurrence of an event of its interest. It specifies the event or events that has occurred. In response to this, the user process can determine its further course of computation.

References

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

[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*, Jan 1993, pp 44-47.

[4] Thomas Gross, David R. O'Hallaron, and Jaspal Subhlok, "Task Parallelism in a High Performance Fortran Framework", *IEEE Parallel and Distributed Technology*, Fall 1994, pp 16-26.

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

[6] *IEEE Parallel and Distributed Technology*, Fall 1994, pp.69

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

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

[9] Rushikesh K. Joshi, D. Janaki Ram, "Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations", *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.

[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.

[14] B. Chapman et al., "Opus: A Coordination Language for Multidisciplinary Applications," *Scientific Programming*, Vol. 6, No. 2, April 1997.

[15] J. Sublhok and B. Yang, "A New Model for Integrated Nested Task and Data Parallel Programming," *Proc. ACM Symp. Principles and Practice of Parallel Programming*, ACM Press , 1996, pp.1-12.

[16] S. Ben Hassen and H. E. Bal, " Integrating Task and Data Parallelism Using Shared Objects," *Proc. 10th ACM Intl. Conf. Supercomputing*, ACM Press, 1996, pp. 317-324.

[17] E. A. West and A. S. Grimshaw, "Braid: Integrating Task and Data Parallelism," *Proc. Frontiers '95: Fifth Symp. Frontiers of Massively Parallel Computation*, IEEE CS Press, 1995, pp. 211-219.