Cooperative Computing for Space Applications using Advanced Instrument Controllers

Dan Watson
Department of Computer Science
Utah State University
Logan, Utah 82322-4205
watson@cs.usu.edu

Ken Blemel
Management Sciences, Inc.
6022 Constitution Ave, N.E.
Albuquerque, New Mexico 87110
ken@mgtsciences.com

Abstract- **This study examines a distributed execution system composed of C51-based miniaturized computing devices called Advanced Instrument Controllers (AICs). Developed by Phillips Laboratories and taking up roughly the space of a postage stamp, AICs combine a digital signal processor with 128K of EEPROM, 128K of RAM, 32 analog input channels, 8 analog channels, and 6 communication ports in a radiation-hardened package. The distributed execution system uses AICs with a task distribution system called MOM, and a planning tool called Generational Scheduling. MOM is a fault-tolerant support system executing across distributed, heterogeneous network of processing elements which has the ability to diagnose faults, identify recovery of nodes, and rollback lost jobs. Generational Scheduling (GS) is simplistic yet effective scheme for dynamically scheduling dependent tasks in a near-optimal manner on available resources. By using AICs, MOM, and GS a self-organizing and self-compensating execution environment is obtained.**

I. INTRODUCTION

In the past and even today, most remote sensing devices gather raw data, and send the data to a centralized processing center. After processing, the data is organized in a format that is useable by scientists. There are a few notable disadvantages with this approach. Often, there is more information than resources at the processing center. Much of the information does not serve to increase knowledge because there is not enough money or time to analyze all of the raw data. Additionally, the ability to communicate the information to the ground requires expensive hardware, requires additional satellite power, and consumes bandwidth. Finally, added computer resources are needed at the data center to process the data into a useable format.

An alternative approach would be for the sensing devices to process the information, analyze it, and communicate results from the experiments, rather than only communicate raw data. This would alleviate many of the aforementioned obstacles by reducing the amount of data communicated to the ground, thereby alleviating the need of additional ground processing hardware, satellite communication power, and bandwidth. The solution should necessarily be small, lightweight, and consume minimal power. Additionally, the solution should be cheap, and preferably bundled into a single product.

Systems that are to perform in a space environment are additionally burdened with a need to provide graceful degradation in the presence of temporal and permanent partial failures, and to have the capability to transition from one phase of a mission to another.

This study examines a distributed execution system composed of C51-based miniaturized computing devices called Advanced Instrument Controllers (AICs). Developed by Phillips Laboratories and taking up roughly the space of a postage stamp, AICs combine a digital signal processor with 128K of EEPROM, 128K of RAM, 32 analog input channels, 8 analog channels, and 6 communication ports in a radiation-hardened package. The distributed execution system uses AICs with a task distribution system called MOM, and a planning tool called Generational Scheduling. MOM is a fault-tolerant support system executing across distributed, heterogeneous network of processing elements which has the ability to diagnose faults, identify recovery of nodes, and rollback lost jobs. Generational Scheduling (GS) is simplistic yet effective scheme for dynamically scheduling dependent tasks in a near-optimal manner on available resources. By using AICs, MOM, and GS a self-organizing and self-compensating execution environment is obtained.

II. AIC: ADVANCED INSTRUMENT CONTROLLER

The target platform for this study is the Advanced Instrument Controller (AIC) [1], developed at the United States Air Force at Phillips Laboratory. The AIC is a unique device that incorporates many features required for monitoring and control of data collection systems. About the size of a large postage stamp, the AIC is a self-contained computer that is based on the Intel 8031 and 8051 microprocessor. It weighs only 5 grams, and consumes less than 50mW. It has I/O ports, RAM, and a programmable EEPROM. The AIC is rugged, where it can operate at ~120 C and withstand 30,000 G-forces. The highly integrated functions allow a reduction in cost, weight, and power, while improving reliability in assembly component interfaces.

The AIC has 32 channels for analog input (with12-bit resolution), that can be used for sensing the environment. Data is stored in RAM during acquisition, and later stored in the EEPROM data section. Furthermore, the analog module has a power reduction mode that is immediately activated by taking the PREN logic line to a low level. Once data is stored in the EEPROM, the AIC may go to low-power mode since low power mode will not preserve any data stored in RAM.

0-7803-5491-5/99/$10.00 © 1999 IEEE

No data acquisition will be done during low power mode. The AIC EEPROM is programmed by holding the PROG logic line high during a hardware reset. This causes the AIC51 to receive serial information from serial synchronous port 0 and program the information in program space of the EEPROM. Reception is accomplished by setting the logic lines SREN = 1 and SRI = 0, which will tri-state the data bus, put all digital outputs to low, gate out all digital inputs, and zero out all DAC outputs.

The AIC requires 2 power supplies, 3.3 volts and 5 volts. Power management is software controlled, and will also control power to the analog module. There are two modes of operation, high and low power mode.

The high power mode allows full operation, with a selectable internal clock speed. The AIC uses an average of 50mW during high power mode. This is always entered at the end of reset or SRAM downloads from EEPROM. This mode is continued until instructed otherwise.

Low Power, or sleep mode, shuts down the clock but keeps oscillator running. The status, and program execution points are still retained. RAM data is lost, so this data must be stored in EEPROM before entering this mode. The AIC used about 70 uW during low power mode. This also assumed there was no load on the output pins. Low Power Mode is exited at the conclusion of a time delay programmed by software into the wake-up circuit prior to initiation. The range is 1 to 4000 seconds, in one second intervals.

At power-up, power is restored and the clock is restarted. It is anticipated that a 9 volt battery will be used for power. The battery should be capable of sustaining the system for several months under normal use.

The AIC has 128K of RAM and 128K of EEPROM. The lower 64K of memory is for program execution. After a hardware reset, the program and data storage are downloaded from EEPROM into RAM. Then, program execution automatically starts from address 0000 from RAM. By setting the AIC register PENB, the EEPROM may be powered down after download.

III. MOM: A RELIABLE DISTRIBUTED EXECUTION ENVIRONMENT

A simple and reliable distributed execution environment is needed. This section provides an overview of one candidate system known as MOM [2], a fault-tolerant support system executing across distributed, heterogeneous network of workstations which has the ability to diagnose faults, identify recovery of nodes, and rollback lost jobs. MOM works on a replicated worker scheme and is modeled on the transaction-oriented paradigm of LINDA [3].

MOM is a middle-ware support system that provides a shared repository of jobs, called as a Tuple Space. Each job is known as a tuple. This Tuple Space is like a bulletin board on which jobs are posted. A tuple may represent a data record that has been produced by a task for consumption by another task or it may represent a task in itself. MOM provides a programming interface to the Tuple Space that allows tasks to read and write tuples on the bulletin board. Conceptually this Tuple Space encompasses the entire system. Physically, it may be implemented to be resident either on a single host in the system or on multiple hosts.

Software systems (i.e. applications) running under MOM are organized as worker tasks running on hosts in the system. Several tasks might be executing on a single host in the system, or if there are enough hosts, each worker task may run on a separate host. Tasks are written using a producer-consumer model; some tasks create new jobs in the form of tuples, while others consume newly created jobs by performing the required work.

One or more supervisor processes maintain the shared repository and monitor the state of the system. These are collectively called the Fault Tolerant Bulletin Board (FTBB) server or MOM server or simply Job server. Under normal operation, workers place and/or grab jobs off the bulletin board on a continuous basis. If one of the tasks "dies", its jobs are restored on the bulletin board by the server, and the state is "rolled-back." Other workers pick up any jobs that were being performed by the (now dead) resource. Additionally, any partial results or jobs posted by the now dead workers are also purged to avoid unnecessary duplication of results and/or work.

In general, MOM is designed to run as a distributed job server, so that all hosts in the system participate in holding and storing jobs. In a system with a full-capability MOM, the loss of multiple supervisors is handled automatically, so that even if multiple supervisors are lost, the remaining supervisors perform all necessary recovery steps.

MOM specifies an application programmer's interface for task to post and grab jobs on the Tuple Space. This interface is very simple and provides five methods:

```
Outputc (tuple)
Input (tuple)
Progress (tuple)
Done (tuple)
Report()
```

A tuple itself has the following structure:

Type:	This describes the type of a tuple, this may be user defined or system specific
Action:	This describes the action to be performed, e.g., `Output, Input, Progress`, etc.
Size:	The size of the user data element contained within the tuple in number of bytes.

Timestamp: The time of day at which this tuple was created. This is used along with a timeout value to determine whether the tuple has timed out.

User-data: This contains the actual user specific data structure that represents work or results of work.

This application-programming interface imposes a paradigm shift on the user programmer. The programmer now needs to dissect the problem into smaller tasks that use a producer-consumer model, similar to the LINDA paradigm. This paradigm is easy to adapt to especially in cases where the design is meant to be parallel and distributed.

In reality the API is implemented as a stub program and compiled into a library, to be linked with the user program, statically or dynamically. The stub does all the communication work with the server. The user programmer simply has to call the interface methods and pass the parameters. Thus for each new platform to be supported for MOM, the stub needs to be ported to that platform. This ensures easy support for new architectures and provides a good degree of support for heterogeneity in the system.

`Outputc()` is used to post new tuples on the bulletin board. `Input()` is used to remove a tuple from the bulletin board. `Done()` is used to indicate that the processing associated with this tuple is complete. `Progress()` is used to reset the timeout clock for the tuple being processed. Report() provides a way to obtain current status of the system.

Consider the following example application program:

```
Tuple t;
t.type = PROCESS;
Input(&t);
if (t != NULL && t.data != NULL) {
     /* Do any processing of t.data here */
     t.type = DISPLAY;
     Output (&t);
}
Report();
```

In this example a single tuple is declared and it is given a desired type. This type (`PROCESS`) is used to match tuples existing on the MOM server, and obtain the matching tuple. The data element contained in the tuple is then extracted and processed. The results are stored in the same memory space after processing. Finally, its type is switched to DISPLAY and the new tuple containing results is then posted back on the bulletin board for other workers to grab and process.

Parallelism can be achieved by replicating a worker task on one or more hosts so that more tuples are processed at the same time. This also ensures fault-tolerance because if a worker of one type dies, another worker of the same type can pick up its lost tuples.

IV. Task Scheduling

Operating systems achieve efficient utilization of precious computing resources through intelligent scheduling of tasks so that the overall completion time is minimized. Strategies for scheduling multiple tasks on a single computing resource (host) exist in all single-host operating systems. Classic solutions for this problem are well established and mostly based on ordered or prioritized assignment of tasks on the single host [4].

In literature, the terms scheduling and mapping have been used extensively and at times alternatively. In reality however, scheduling is different from mapping in the sense that essentially, mapping is a part of the scheduling process. In this study the term mapping refers to the portion of the scheduling process during which the scheduler assigns a task (or a set of tasks) to an available and eligible host (or a set of available and eligible hosts). In other words, it can also be seen as a process during which the most appropriate host is "selected" for assignment. The mapping algorithm referred to in this chapter does not allocate or delegate tasks to selected or assigned host(s).

Scheduling is said to be static when all the tasks to be scheduled and all the hosts that are available and eligible to select are known before scheduling begins. In addition, there may be several other pieces of information necessary to be known prior to scheduling the tasks. These will be discussed in detail in the sections to follow. Scheduling is said to be dynamic when tasks are submitted during run-time for scheduling, and the scheduler has a much more formidable task of trying to assign these tasks to hosts within tight time-constraints.

The problem of finding an optimal matching of n tasks to m hosts is NP hard. The effort required for discovering this matching varies greatly with the nature of tasks to be completed and the structure of the system environment. The criterion for determining an efficient schedule also varies from system to system. More often than not, the overall completion time of a schedule is considered as the determining factor. The overall completion time for a set of scheduled tasks is the time when the last task scheduled is completed. Scheduling may be constrained by factors such as data dependencies between tasks, availability of hosts, variable communication capabilities between hosts, and differences between computing resources required by the task and those offered by the host. In this discussion a host that satisfies these constraints is called an available-eligible host. Thus, the problem of HC scheduling is the problem of finding an optimal schedule and matching of the given set of tasks for execution on the given set of hosts such that:

a) Overall completion time is minimized
b) Scheduling constraints are satisfied.

Such a definition of the problem inherently makes it a static-scheduling problem as it is assumed that a set of tasks is known before scheduling commences. An example of dynamic scheduling is provided later in the document.

In order to help define the scheduling problem, the following assumptions are made:

a) All hosts in the set of given hosts are available-eligible for assignment.
b) The estimated times for a task to complete on each of the possible hosts is known.
c) The estimated rates of data transfer between given hosts are known.
d) The data dependencies between tasks are known.

Additionally, the following elements are known:

- A set of n tasks which are interdependent and may require transfer of data between them, denoted as the set $T = \{ t_1, t_2, \ldots, t_n \}$.
- A set of m hosts which are available-eligible for scheduling the set T of tasks, denoted as the set $H = \{ h_1, h_2, \ldots, h_n \}$.
- An $n \times m$ matrix of *ETC* (Estimated Time to Complete) values such that *ETC[i][j]* is the estimated time to complete task t_i on host h_j. (e.g., [5]).
- A task dependency graph, i.e. a directed acyclic graph G that indicates data dependencies between tasks from T. The set of vertices V represents tasks from T and the set of edges E represents the precedence relations between them. (Figure 1). An edge from node i to node j indicates that execution of task t_i must precede execution of task t_j. Edges are weighted with values that represent amount of data transfer necessary between dependent tasks.
- An $m \times m$ matrix of transfer *rate* values such that *rate[i][j]* is the average rate of transfer of data between hosts h_i and h_j.

The scheduling problem is then to find an assignment from T to H, $S(t_i)$, such that the *completion time* of task t_f, the last task in T to finish, executed on host h_j, is minimized.

Generational Scheduling (GS) [5] operates by scheduling and rescheduling only those tasks that have all of their dependency constraints met. GS has been found to provide near-optimal schedules for most scheduling problems. It has four separate components.

First, GS forms a scheduling problem composed of all the tasks known to exist, the dependencies between those tasks, and the time needed to execute those tasks on each of the processors given their current loading.

Second, GS filters out all tasks except those that have all their dependency constraints met (i.e., they are ready to execute). GS does not consider any task that is not ready to run, or any task that is currently executing (i.e., no preemption).

Third, GS uses a fast, simple scheduling algorithm to schedule the reduced problem. A scheduler that is O(elegible_tasks*hosts) is sufficient. Note that a task is reconsidered for execution on a different host each time this algorithm is run.

Finally, a mechanism is needed to detect events in the system that would warrant a re-formation of the entire problem (i.e., rescheduling events). Typical rescheduling events include the completion of a task, or the arrival f a new task.

GS can be readily implemented as a dynamic scheduling tool, because the partial solution is immediately implemented (i.e. jobs are executed before the entire solution is determined). The scheduler is simply called again as tasks finish execution. Existing scheduling algorithms can be employed and newer algorithms can be incorporated as they are developed. Furthermore, the dynamic version is *easier* to implement than static version, because rescheduling event detection mechanisms are already typically in-place, and there is no need for sophisticated schedule monitor/control software.

REFERENCES

[1] *AIC Users Guide*, July 8 1997, Mission Research Corporation, (technical manual).

[2] Cannon, S., and Brinkerhof, D., "A stable distributed tuple space," *Proceedings of International Conference on Systems Sciences*, Jan 1996.

[3] Cannon, S., and Dunn, D., "Adding fault-tolerant transaction processing to LINDA," *Software: Practice and Experience*, Vol. 25, No. 5, 1994, pp. 449-466.

[4] Fernandez-Baca, D., "Allocating modules to processors in a distributed system," *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 11, November 1989, pp. 1427-1436.

[5] Carter, B., R., Watson, D., Freund, R. F., Keith, E., Mirabile, F., and Siegel, H. J., "Generational scheduling for dynamic task management in heterogeneous computing systems," *Proceedings of the International Conference on PDPTA*, June 1996, pp. 769-778.