Implementation of Farming Under PVM

Sameer S. Shende
Department of Computer and Information Science
University of Oregon

April 25, 1995

Abstract

Dynamic load balancing can improve the performance of a parallel application significantly. This is needed in the PVM environment because different nodes in the network have different processing capacities, networks have different data carrying capacities, in a timesharing environment, other users' processes can slow down a particular task in a parallel application. Also a large number of data parallel algorithms require unequal processing times for each quantum of data given to the processor.

This paper explains the design and implementation of one such load balancing scheme under PVM. PVM is fast becoming a popular parallel programming environment both on a network of workstations, multiprocessors and other parallel and vector platforms.

Keywords : dynamic load balancing, PVM, farming, heterogeneous computing, parallel and distributed processing.

1 Introduction

This paper focuses on an implementation of process farms under the parallel virtual machine environment. PVM uses a cluster of workstations to form a heterogenous network based MIMD computing environment. The problems associated with such an environment to form a parallel application are that, typically point to point communication constructs are provided to distribute data in a data parallel approach. Most parallel applications need some form of load balancing technique to equitably distribute work packets and keep all processors busy. This is usually a static off line process in which the application is evaluated for a particular problem size and a log is examined to determine the placement of tasks on the nodes by examining the computation to communication time ratios for the tasks. The application has to explicitly distribute the data so as to have equal load on all processors.

2 Load Balancing

2.1 Need for Dynamic Load Balancing under PVM

In PVM, the user tasks which comprise a parallel application are distributed over the nodes in a network. The nodes could be parallel/vector machines or workstations. Thus, a heterogenous computing environment is used to solve the problem. To extract the maximum performance from the parallel application, it is important to keep the idle time of tasks to a barest minimum. To ensure that each process gets an optimum load to be processed, a dynamic load balancing scheme is desirable. The relevant issues that the application developer has to take into consideration are :

1. All nodes in the network do not have identical computation capacities. Even among workstation class machines, a variety of processor architectures, network interfaces, and I/O subsystems can be encountered. Some machines may have processing rates which are an order of magnitude higher than some other machines in the network.

2. A parallel application running over a wide area network will encompass a range of networks which have different transfer rates and latencies. For e.g., ATM supports 622 Mb/s and 100 Mb/s Ethernet LANs have 10 Mb/s rates, WANs have 64 Kb/s (X.25), Token ring LANs have 16 Mb/s peak transfer rates. This results in different message transfer rates between tasks in the parallel application.

3. There are background jobs in most time sharing multiuser systems (typical unix workstations) so the CPU time that a task under PVM can get may vary from one run to another depending on the usage of the workstation.

4. A large number of algorithms are not structured nicely, and on breaking the data to be processed into work packets, each work packet may require a different processing time (even if we keep the above factors constant). This unequal processing time requirement cannot be always determined apriori and can be difficult to account for while designing the application.

2.2 Dynamic Load Balancing

Dynamic load balancing in an application can address these concerns. Dynamic load balancing can be implemented in a heterogenous environment either at the operating system level or at a process level.

1. At the *operating system level*, the emphasis is to keep all the *processors* busy all the time. This is accomplished by process migration from a heavily loaded node in the network to a lightly loaded one. This raises issues about doing this effectively at runtime on a heterogenous collection of nodes. While it may be easy to incorporate such a scheme for a “state-less” server process by killing one instance and starting another

instance on the other node (different binaries being used for dissimilar processors and operating systems), it can be difficult to migrate a process at run-time when the the state of a process needs to be maintained. The process migration approach can be seen in systems such as Condor.

2. At the *process level*, it is important to keep all the *user tasks* busy all the time. If the data to be processed is divided into independent work packets, then the load balancing scheme at the process level can be implemented by giving more work packets to the tasks on faster processors and less work packets to the slower ones. This way, the data can be distributed on the basis of the processing capacity of a particular task. This approach takes into account the differences in processing times for different data packets. The aim of the paper is to implement a dynamic load balancing scheme under PVM using this approach by constructing *process farms*.

3 Farming

3.1 Farming - Definition of terms

Under the farming model, the parallel application is divided into one or more process farms. A process farm comprises one farmer task and one or more worker classes. A worker class has one or more worker tasks which are grouped together because they perform similar functions. A worker (task) can be associated with one and only one farmer task (and in turn one worker class). In a *hierarchical farm*, a worker task can in turn be a farmer to some other set of worker classes and worker tasks. Again, the worker can be a farmer to more than one farm. This could be useful in subdivision of a work packet into sub parts for processing by another worker class hierarchy. Also, the farm implementation should be such that a worker task should be free to leave a process farm (dynamic process migration to another farm) and join any other farm if needed.

To implement an algorithm using the farming paradigm, the data parallel application would need to create one or more farms. The farmer task would need to take the data to be processed and split it up into a number of work packets which would be independent of each other and could be processed by a worker task. It would then give out the work packets to a worker class without knowing which particular worker task would process it. The operating environment would decide the appropriate worker to forward the work packet to. After processing the work packet, the worker would create a reply packet and address it to the farmer task. The farmer task would collect all the work packets from a particular worker class and reassemble these appropriately. More than one farm and more than one worker class could be active at a particular instant creating, sending or receiving packets. The farmer could terminate a farm after it is no longer needed, at which stage the worker tasks would automatically be expelled from the farm and informed accordingly. However, before this happens, a worker should also be able to leave the farm and stop receiving work packets from the worker class.

3.2 Farming - Classification

To focus our attention on the operating environment (or a farm daemon task), it should be able to manage the creation of the farms and operations such as termination of a farm, leaving a farm (by a worker task), and routing of work packets to the worker. There are ways in which the packet distribution to a worker can occur. Farms are classifed into the following categories.

1. *Static Farm*

   In this case, the farm daemon could distribute all the work packets, that it has at any instant, equally to all the worker tasks. The advantage of this scheme is that the worker task would not need to request the farm daemon for a work packet but would retrieve it locally from its queue of messages. The disadvantage of this scheme is that at run-time if another worker task joins a worker class after all work packets have been distributed, then a redistribution of work packets would not take place and the new worker task could be idle. Also, this approach does not take into account the different processing requirements for each work packet. So, in this approach the farm is static in the sense that all worker tasks can process number of packets that is independent of their processing capacities. So this does not provide for dynamic load balancing.

2. *Dynamic Farm*

   In this case, the farm daemon would distribute the work packets to workers only when the worker would request for a work packet. Thus, different workers could receive different number of work packets depending upon their respective processing capacities. Also, this scheme is suitable to a farm which allows migration of a worker task from one farm to another. When this happens, the new worker task could receive a work packet as all the work packets are contained in the central farm daemon task and it can be integrated into the farm easily. The advantage of the dynamic farm is that it can support worker task migration (between farms) efficiently, it can distribute work packets depending upon the processing capacities of the worker tasks and keep all the workers busy most of the times and so it implements a dynamic load balancing scheme. The disadvantage of a dynamic farm is that it requires the work packets for a farm to be stored at a central location which requires a large storage to be allocated at the farm daemon. Also, the worker tasks need to request for each work packet explicitly, so there is a greater latency in getting a work packet as compared to the static farm case where all the work packets are locally stored and can be retrieved without any additional messages to be sent to the central server (farm daemon).

   Different distributed data management schemes with respect to locks are discussed in [?] . These schemes are analogous to managing the work packets in the case of process farms.

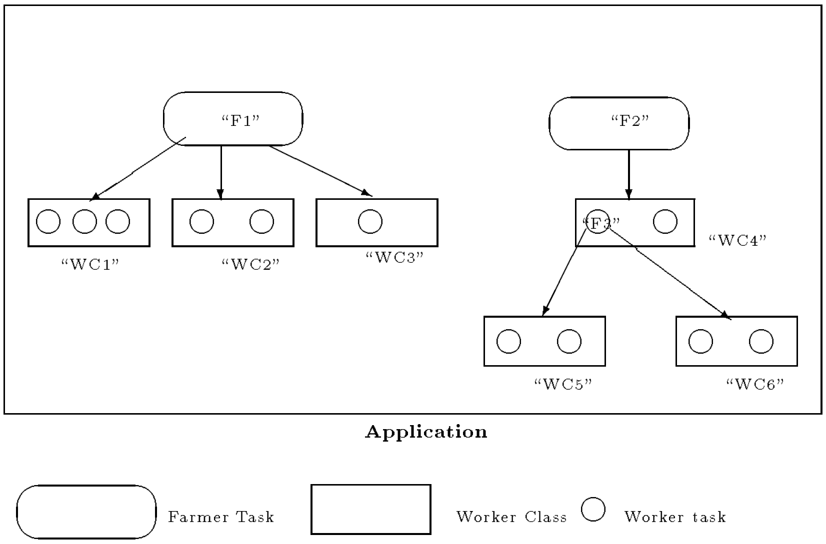

Figure 1: Representation of Farming

3.3 Goals of the farming implementation

The aim of this paper was to implement a hierarchical farm which supported worker task migration from one farm to another and to implement a dynamic load balancing scheme by using a dynamic farm model described above. It needed to support multiple farms at each level with multiple worker classes at each farm level. Furthermore, it was necessary to formulate an Application Programmers Interface (API) for the farm which was independent of the nature of the farm (static or dynamic farm implementation) and would work with the PVM environment. It was important to design it in such a way that there would be *"busy-waiting" condition* at any stage in the components. Any polling for an event to happen would only degrade the system performance. Also, it was a goal to use the farming constructs as a layer above the PVM software and to use the interface of PVM to build the farming model. This was necessary to support the farm implementation across different versions of PVM (starting from 3.2.0) so that it would give backward source code compatibilty (as long as PVM maintained backward compatibility).

3.4 Previous work in this area - Comparison

Work related to the implementation of process farms has been done on transputer based environments which are typically distributed memory MIMD message passing environments. The work done at EPCC [?] on process farming runs on top of their CHIMP environment which also works as a heterogenous computing platform using workstation clusters. Their implementation of farming however consists of a source process and a sink process (instead of a single farmer task). Implementations of static farms with queue lengths (of the local worker task's queue of work packets) specified by worker tasks have been done at C-DAC, India [?] and TUL, India [?] but have concentrated on a homogeneous environment. The advantage of

[?] over this paper's implementation is that the farm related information is maintained in a distributed manner, so when multiple farms are implemented it would have no bottlenecks at the central server site. This may happen in our case if all farms simultaneously dispatch work packets to the workers. Work on a single process farm (non hierarchical) based application implementation also appears in 3L[?] in Inmos, U.K.

The author is not aware of any process farm implementation under the PVM heterogenous computing environment to implement dynamic load balancing. This work aims at bridging this gap by providing a dynamic load balancing construct to expand the PVM API.

4 Implementation and Design Issues

The implementation of process farms consisted of

- Defining the API for farming.
- Designing the software components viz., the farm daemon (pvmfarmd - the server task) and the library of the API (libfarm.a - the client) which would be used by both the farmer and the worker tasks.
- Defining the protocol for interprocess communication between the farmer task and the farm daemon task, and between the worker task and the farm daemon task so as to avoid any polling by any of the three components.

4.1 API definition

At the farmer end:

1. **int pvm_farm_init (char *farm_name);**

   The routine pvm_farm_init() creates a process farm with a name specified by the argument *farm_name*. In an application, the farm name is unique (i.e., the user should not invoke this routine twice with the same name). Multiple farms can be used with a separate farm names. This name is used by the worker task to enroll in a farm. Returns OK or ERROR or PvmFarmNamePresent.

   ```
   e.g.,
   pvm_farm_init("Farmer1");
   ```

2. **int pvm_get_worker_class_id (char *farm_name, char *worker_class_name);**

The routine pvm_get_worker_class_id() returns the identifier that represents a worker class named *worker_class_name* associated with the farm named *farm_name*. This is used by the farmer to uniquely identify a particular worker class. Returns ERROR on error or a positive integer *id*. This identifier is used by subsequent farmer calls to send data packets to this worker class or to receive replies from the worker class. This call blocks till the worker class is created by a worker task.

```
e.g.,
id = pvm_get_worker_class_id("Farmer1", "WC1");
if (id == ERROR)
{
printf("Error in getting worker class id\n");
pvm_exit();
exit(1);
}
```

3. **int pvm_send_work_packet(int id);**

   The routine pvm_send_work_packet() sends the message to one worker task associated with the worker class whose identifier is specified by *id*. The data should be packed in a buffer (the active send buffer) created by pvm_initsend() or pvm_mkbuf(); PvmDataDefault is used to create the buffer so that XDR encoding can be done for communication between dissimilar hosts (little endian and big endian hosts). Returns OK or ERROR.

```
e.g.,
int id, data, bufid;
id = pvm_get_worker_class_id("Farmer1","WC1");
bufid = pvm_mkbuf(PvmDataDefault); /* pvm call creates buffer */
pvm_setsbuf(bufid); /* sets this as the active send buffer
This is also the default action taken */
data = 500; /* gets data ready for packing in the buffer */
pvm_pkint(&data, 1, 1); /* packs one int data item, stride = 1 */
/* calls to pack the data */
pvm_send_work_packet(id, 34);
pvm_freebuf(bufid); /* frees the buffer */
```

4. **int bufid = pvm_recv_reply_packet(id);**

   This blocking receive reply packet routine will wait until has arrived from the worker class specified by the identifier *id*. This call places the message in a new active receive buffer that is created. The previous active receive buffer is cleared unless it has been saved explicitly with a pvm_setrbuf() call.

```
e.g.,
pvm_recv_reply_packet(id);
```

5. **int pvm_farm_terminate(char *farm_name);**

   This routine shuts down a farm and frees the worker tasks in the worker classes associated with the farm whose name is specified by *farm_name*. The worker tasks waiting on a receive are sent a message that the farm has terminated. After executing this call, a new farm with the same name may be created. Returns OK or ERROR or PvmNoSuchFarm or PvmNotFarmOwner.

   ```
   e.g.,
   pvm_farm_terminate("Farmer1");
   ```

6. **int pvm_start_farmd(void);**

   This routine starts the pvmfarmd daemon which is responsible for farming related calls. This daemon can be started manually from the pvm shell or by using this call. This should be called only once in the entire application. Returns OK or ERROR.

   ```
   e.g.,
   pvm_start_farmd(); /* starts the daemon.*/
   ```

7. **int pvm_stop_farmd(void);**

   This routine sends a message to the pvmfarmd to terminate itself. This should also be called by only one task in the application. This should be done by the farmer task after it has terminated the farm. Returns OK or ERROR.

   ```
   e.g.,
   pvm_stops_farmd(); /* stops the daemon.*/
   ```

   At the worker end:

8. **int pvm_init_worker_class(char *farm_name, char *worker_class_name);**

   The pvm_init_worker_class() routine creates (if it already doesn't exist) a worker class named *worker_class_name* and enrolls the calling task in the worker class. This worker class is associated with the farmer task specified by *farm_name*. This call blocks till a farm is created (in case the farmer has not created the farm). Returns OK or ERROR.

   ```
   e.g.,
   pvm_init_worker_class("Farmer1", "WC1");
   ```

9. **int bufid = pvm_recv_work_packet(void);**

   The pvm_recv_work_packet() routine waits until a message arrives from the farmer task. A worker task can be associated with at most one farmer (and a worker class) at a time. The previous active receive buffer is cleared unless it has been saved with a pvm_setrbuf() call. This call returns bufid or PvmFarmTerminated if the farmer has executed a farm_terminate() call. When PvmFarmHasTerminated is returned, the worker task is free to do any other activity not associated with the current farm.

   ```
   e.g.,
   buffer_id = pvm_recv_reply_packet();
   /* All unpacking routines */
   ```

10. **int pvm_send_reply_packet(void);**

    This routine sends a reply packet which has been packed into the active send buffer, to the farmer task. Returns OK or ERROR.

    ```
    e.g.,
    /* initialise the buffer for sending */
    int bufid,data;
    bufid = pvm_mkbuf(PvmDataDefault); /* makes a new buffer */
    pvm_setsbuf(bufid); /* done by default - sets this
    as the active send buffer used by pvm_send_reply_packet() */
    data = 5; /* some reply packet */
    pvm_pkint(&data, 1, 1); /* packs the integer into the buffer */
    pvm_send_reply_packet(); /* sends the buffer to the farmer */
    pvm_freebuf(bufid);
    ```

11. **int pvm_leave_farm(void);**

    This routine is called by the worker task to disassociate itself from the farm in which it is currently enrolled. This call removes the task related information of this task from the worker class information repository maintained by the central pvmfarmd daemon (farm daemon). After this call is executed, the calling worker task no longer receives work packets from the farmer. The worker can enroll in any farm subsequently, by calling the pvm_init_worker_class() routine. This allows for the worker process to migrate from one farm to another dynamically at runtime. Returns OK or ERROR.

    ```
    e.g.,
    pvm_leave_farm(); /* leaves the farm */
    ```

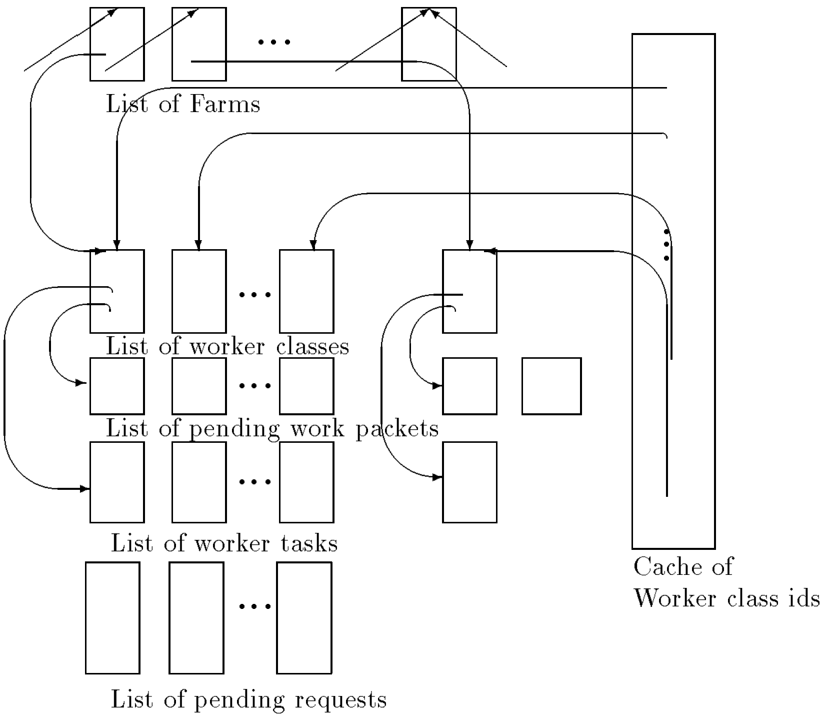

Figure 2: pvmfarmd - Internals

4.2 Design issues

The process farms were implemented following the design of the farm daemon and the definition of the protocol for interprocess communication as described in Appendix A of the paper. The farm daemon (pvmfarmd) was implemented using lists. The entire code was written in C++ making list as a class with member functions like append, delete, pop, search by name, search by value, etc. The farm daemon had a list of all farms (each farm was identified by a unique name and farmer task id). Each item in the farm list had a list of worker classes that were associated with it. Each item in the list of worker classes had a unique worker class identifier, the list of work packets awaiting distribution and a list of worker tasks enrolled in the worker class. Apart from these lists, there was a global list of all pending requests (e.g., the farmer requested for a worker class id and the worker class was not created, so it had to wait for a worker class to be created, etc.). When a work packet arrived at the farm daemon, it had to be forwarded to a single worker task. To search for the worker class to which the packet had to be addressed, one approach was to search each item in the farmer list for a matching farm name, then to get to the worker class list and then to search for the matching worker class id item and then to get a worker task that had no pending work packets. This was inefficient when a large number of work packets were sent to the farm daemon. Thus, to improve performance, a cache of worker class ids was maintained. The entries in this cache array pointed to the worker class item in the list of worker classes. So, in addition to the lists, the array of worker class items (indexed by the unique worker class