Appendix A

CARMI User's Guide

A.1 Submitting CARMI jobs via Condor

Submitting a job to CARMI is much the same as submitting any other job to Condor. The biggest difference is the requirement that the Condor job universe must be set to PVM. Second, the number of machines desired must be specified. This is done with a `machine_count` statement. A `machine_count` is of the form: `machine_count = n..m` where `n` is the lower bound on number of machines required, and `m` is the upper bound on the machines desired. A job which can use as many machines as it can get can simply specify an extremely large upper bound. A sample CARMI submit file is shown in figure 29.

This submission file assumes the job can use between one and five machines of the same type as the one the submission command is run on. Getting machines of a type other than the submitting machine requires explicitly writing

```
universe = PVM
executable = carmi_program
arguments = -foo -bar 2
output = out_file
error = err_file
# This job needs between 1 and 5 machines.
machine_count = 1..5
queue
```

Figure 29: Sample Condor submission file for a CARMI job

a requirements expression in the submission file, prior to the queue statement.

One job can request machines of different types by alternating requirements and queue statements as in figure 30. In this example, Condor will allocate between one and three workstations running Ultrix4.3 to the job, exactly four workstations running HPUX, and between one and one hundred machines running SunOS to the job prior to starting the job. Each requirements expression defines another CARMI resource class. These classes are named by number, starting with "0" for the first `requirements` statement, and increasing by one for each queue statement. Therefore, the Ultrix machines will be in CARMI class "0," the HPUX machines in CARMI class "1," and the SunOS machines in class "2." As will be seen later, CARMI provides services for defining new classes, and retrieving these class definitions at run-time.

```
machine_count = 1..3
requirements = OpSys == "Ultrix4.3" \&\& Arch == "MIPS"
queue
machine_count = 4..4
requirements = OpSys == "HPUX9" \&\& Arch == "HPPAR"
queue
machine_count = 1..100
requirements = OpSys == "SunOS4.1.3" \&\& Arch == "sun4m"
queue
```

Figure 30: Submitting a Multi-class CARMI job

A.2 Differences in PVM under CARMI

CARMI uses PVM for the basic set of application communication primitives. Therefore, there are no changes in how PVM application processes communicate when running under CARMI. CARMI also supports the PVM resource management API (e.g. `pvm_spawn()` with a few important exceptions.

First, where ever PVM uses an architecture name (for example "SUN4"), CARMI replaces it with a class name. Any place PVM uses an architecture as a parameter (e.g. `pvm_addhosts()`), or returns it in a function call (e.g. `pvm_config()`), CARMI replaces the architecture with a class name.

CARMI also does not use the directory structure normally used by PVM for storing binaries. CARMI assumes that all binaries are stored in the same directory from which the job was submitted. Therefore, any processes to be created via the `pvm_spawn()` function should have their binaries stored in the same directory from which the user submits the CARMI job to Condor.

A.3 CARMI API

The CARMI API adheres to the request protocol set-up by our resource management framework [39]. That implies that every CARMI function immediately returns a request identifier, and that every function requires an argument which specifies the tag to be used on the request response message. The function definitions below are followed by a returns(...) which specifies the data types in the response message. All of these messages are assumed to begin with a request identifier field, which is not shown here.

A.3.1 Request Management

```
RequestId  carmi_cancel_request(RequestId id,
            ResponseTag resp_tag)
returns(BOOLEAN success)
```

carmi_cancel_request the outstanding request with id. The response message returns false if no request with the given id is outstanding. The side effects of canceling a request are dependent on the type of request which has been deleted.

A.3.2 Resource Handling

```
RequestId  carmi_config( ResponseTag resp_tag)
returns(int nhost, HostId hosts[nhost])
```

carmi_config returns the number of hosts currently available to the application, and a list containing the host identifiers.

```
RequestId  carmi_get_class_ad(HostId id, ResponseTag resp_tag)
returns(ClassAd machine_ad)
```

This function returns all available information concerning a host. The type "ClassAd" is borrowed from Condor, and it includes information which characterizes a particular resource. The ClassAd structure is extensible, but the minimum content includes attributes such as processor and operating system type. A ClassAd is a C++ object, and includes methods for discovering all the attributes stored in the ad, and retrieving the value associated with any attribute.

```
RequestId  carmi_addhosts(char *class_name, int count,
             ResponseTag resp_tag)
returns(int count, HostId new_hosts[count])
```

carmi_addhosts requests new hosts in class class_name. Hosts will be added, one at a time, until count total have been added. Multiple response messages to the addhosts functions may be generated until the sum of the count in all of the responses equals the count in the initial request.

```
RequestId  carmi_delhosts(int count, HostId hosts[],
             ResponseTag resp_tag)
returns(int count, HostId host_id[count])
```

This function requests that hosts be deallocated from the application. The response message contains the host identifiers of the machines successfully removed. In a successful execution, this list will exactly match the list given in the request.

```
RequestId  carmi_add_notify(char *class_name,
             ResponseTag resp_tag)
returns(SAME as addhosts above)
```

carmi_add_notify requests that messages be sent to the application when new hosts are added to the system due to the original set of requests in the submission file (as opposed to a run-time addhosts() request). The response messages are the same as those returned in response to an addhosts request. This request is outstanding until explicitly canceled (though after all resources specified by the submit file have been granted it will never be triggered).

```
RequestId  carmi_delete_notify(int count, HostId host_ids[],
             ResponseTag resp_tag)
returns(SAME as delhosts above)
```

carmi_delete_notify requests that messages be sent to the application when a host is lost due to an event other than an explicit carmi_delhosts() request (e.g. hardware failure, or host being reclaimed by its owner). The format of these notification messages is the same as the response to a carmi_delhosts() request.

```
RequestId  carmi_{suspend, resume}_notify(int count, HostId
             host_ids[], ResponseTag resp_tag)
```

```
returns(int count, HostId host_ids[count])
```

These two functions request that the application be notified whenever a host in the specified host list is suspended or resumed by Condor.

A.3.3 Task Management

```
RequestId  carmi_class_spawn(char *executable, char **argv, char
             *class_name, ResponseTag resp_tag)
returns(ProcessId id)
on process exit:  returns(ProcessId id, int status, int
CPU_usage)
```

carmi_class_spawn triggers the creation of a new process. The new process will be started on a host within the class specified. When the process has successfully been created, a response message containing the identifier of the next task is returned. When that process exits, the application receives a message containing exit status information and processor utilization.

```
RequestId  carmi_process_info( ResponseTag resp_tag)
returns(int nprocs, struct procinfo proc_list[nprocs])
```

carmi_process_info returns information on the process running in the system. Information included in the procinfo structure includes the ProcessId, the HostId where the process is running, the ProcessId of the process' parent, and the name of the executable file being run.

A.3.4 Class Management

```
RequestId  carmi_class_definitions( ResponseTag resp_tag)
returns(int count, ResourceClass list[count])
```

This returns the definition of all existing classes. A ResourceClass definition is a string containing an expression which defines the characteristics a resource must have to be considered a member of a class. The string will be the same as the requirements expression in the submit file, or provided to carmi_define_class described below.

```
RequestId  carmi_define_class( ResourceClass,
             ResponseTag resp_tag)
returns(BOOLEAN success)
```

carmi_define_class defines a new resource class, but does not request that any hosts in this class be added. The return value is false if a class with the same name already exists. The string provided in the ResourceClass parameter is of the same form as a requirements expression in the Condor submit file.

```
RequestId  carmi_remove_class( char *class_name,
             ResponseTag resp_tag)
returns(BOOLEAN success)
```

This function removes the class with the given name. carmi_remove_class fails if any hosts in this class are currently allocated to the application. Therefore, all hosts in the class must be deallocated via carmi_delhosts prior to calling carmi_remove_class.

A.3.5 Checkpointing

```
RequestId  carmi_ckpt(int ckpt_cnt, ProcessId *ckpt_list, int
               consistent_cnt, ProcessId *consistent_list, int
               flush_cnt, ProcessId *flush_list)
returns(BOOLEAN success)
```

This function provides a general interface to checkpointing services provided by CoCheck. carmi_ckpt specifies three groups of processes. The first group has ckpt_cnt members with task identifiers in the vector ckpt_list. The processes in this group are checkpointed. The next group, specified by consistent_cnt and consistent_list will be suspended until the processes in ckpt_list have been re-started. Because they are held suspended, CoCheck is able to install communication identifier maps in the processes, so that they will continue to use the same communication identifiers after the checkpoint as they did before the checkpoint. The last, flush, group will simply have the network cleared of any messages in transit to the checkpointing processes. These processes will continue immediately, but will not be able to continue to use the same communication identifiers for the processes in ckpt_list following a restart.

```
RequestId  carmi_restart(int restart_cnt, ProcessId *restart_list,
               int consistent_cnt, ProcessId *consistent_list)
returns(int count, ProcessId[count])
```

carmi_restart is used to restart processes checkpointed previously via carmi_ckpt. The restart_list specifies the processes to be re-started. Typically, these will be the same as the ckpt_list group in the call to carmi_ckpt. The processes in consistent_list are assumed to be blocked waiting for these processes to restart as a result of being in the consistent_list of a previous call to carmi_ckpt.

```
RequestId  carmi_migrate(ProcessId target, HostId host, int
               consistent_cnt, ProcessId *consistent_list)
returns (ProcessId new_id)
```

This function performs a migration of the process specified by target to the machine specified by host. If the machine is no longer available, or of a different class than where target is initially running, an error is returned. The processes in consistent_list will be blocked during the duration of the migration, and will be able to continue communicating with the migrated process using the identifier specified in target. A new ProcessId is returned which can also be used for communicating with the migrated process.