CRL version 1.0
User Documentation

Kirk L. Johnson, Joseph Adler, and Sandeep K. Gupta*
MIT Laboratory for Computer Science
Cambridge, Massachusetts 02139, U.S.A.

24 August 1995

1 Purpose of This Document

The purpose of this document is to aid the user in understanding how to build and use version 1.0 of the *C Region Library* (CRL) [2]. It provides a brief outline of the CRL programming model, describes the CRL commands, gives a short tutorial on writing CRL programs, and explains how to obtain and compile CRL. This version of CRL is still primarily a research vehicle; it is not intended to be of production quality. CRL 1.0 includes support for the MIT Alewife Machine [1], Thinking Machine's CM-5 [3] using active messages (as provided by CMMD [6]), and networks of Sun workstations (running SunOS 4.1.3) communicating over TCP-based local area networks. (For simplicity, we refer to the implementation of CRL for networks of Sun workstations as the TCP/UNIX implementation.) A "null" CRL implementation that provides null or identity macros for all CRL functions except `rgn_create` (which is a simple wrapper around malloc) is also available to obtain sequential timings on Alewife, the CM-5, or uniprocessor systems (*e.g.,* desktop workstations).

To use the TCP/UNIX implementation of CRL, PVM [5] must also be installed on each machine; CRL leverages off of the group and process management facilities provided by PVM. The TCP/UNIX implementation included in CRL 1.0 is not intended to be a high-performance distributed shared memory implementation delivering speedups similar to those obtained with the Alewife and the CM-5 implementations. It is provided to allow development and experimentation by those without access to an Alewife or CM-5 platform and to display the ease with which CRL may be ported.

The rest of this document proceeds as follows. Section 2 provides an introduction to the CRL distributed shared memory system and its goals. Section 3 describes the CRL programming model and CRL functions. Section 4 mentions some of the caveats to keep in mind while using CRL 1.0. Sections 5, 6, and 7 provide instructions for using the CM-5, TCP/UNIX, and null versions of CRL 1.0, respectively. Section 8 gives instructions for obtaining the most recent CRL distribution package. Section 9 contains the copyright notice for CRL. Finally, Appendix A contains a detailed example of a simple application written using CRL 1.0.

*E-mail: `crl-info@pdos.lcs.mit.edu`; WWW: `http://www.pdos.lcs.mit.edu/crl/`

2 Introduction to CRL

The *C Region Library* (CRL) is an all-software distributed shared memory (DSM) system intended for use on message-passing multicomputers and distributed systems. Parallel applications built on top of CRL share data through *regions*. Each region is an arbitrarily sized, contiguous area of memory. The programmer defines regions, and includes annotations to delimit accesses to those regions. Regions are cached in the local memories of processors; cached copies are kept consistent using a directory-based coherence protocol. Because coherence is provided at the granularity of regions instead of memory pages, cache lines, or some other arbitrarily chosen fixed-size unit, CRL avoids the concomitant problems of false sharing for coherence units that are too large or inefficient use of bandwidth for coherence units that are too small.

Three key features distinguish CRL from other software DSM systems. First, CRL is system- and language-independent. Providing CRL functionality in programming languages other than C should require little work. Second, CRL is portable. By employing a region-based approach, CRL is implemented entirely as a library and requires no functionality from the underlying hardware, compiler, or operating system beyond that necessary to send and receive messages. Third, CRL is efficient. Very little software overhead is interposed between applications and the underlying message-passing mechanisms. While these features have occurred in isolation or in tandem in other software DSM systems, CRL is the first software DSM system to provide all three in a simple, coherent package.

Because *shared address space* or *shared memory* programming environments like CRL provide a uniform model for accessing all shared data, whether local or remote, they are relatively easy to use. In contrast, message-passing environments burden programmers with the task of orchestrating all interprocessor communication and synchronization through explicit message passing. While such coordination can be managed without adversely affecting performance for relatively simple applications (*e.g.,* those that communicate infrequently or have relatively simple communication patterns), the task can be far more difficult for large, complex applications, particularly those in which data is shared at a fine granularity or according to irregular, dynamic communication patterns.

In spite of this fact, message passing environments such as PVM and MPI are often the *de facto* standards for programming multicomputers and networks of workstations. We believe that this is primarily due to the fact that these systems require no special hardware, compiler, or operating system support, thus enabling them to run entirely at user level on unmodified, "stock" systems. Because CRL also requires minimal support from the underlying system, it should be equally portable and easy to run on different platforms. As such, we believe that CRL could serve as an excellent vehicle for applications requiring more expressive programming environments than those provided by PVM or MPI.

2.1 Goals of CRL

Several major goals guided the development of CRL. First and foremost, we strove to preserve the essential "feel" of the shared memory programming model without requiring undue limitations on language features or, worse, an entirely new language. In particular, we were interested in preserving the uniform access model for shared data, whether local or remote, that most DSM systems have in common. Second, we were interested in a system that could be implemented efficiently in an all-software context and thus minimized what functionality was required from the underlying hardware

and operating system. Systems that take advantage of more complex hardware or operating system functionality (*e.g.,* page-based mostly software DSM systems) are worthy of study, but can suffer a performance penalty because of inefficient interfaces for accessing such features. Finally, we wanted a system that would be amenable to simple and lean implementations in which only a small amount of software overhead sits between applications and the message-passing infrastructure used for communication.

3 Programming with CRL

Applications using CRL can effect interprocessor communication and synchronization either through a relatively typical set of global operations (barrier, broadcast, reduce) or through operations on *regions*.

A region is an arbitrarily sized, contiguous area of memory identified by a unique *region identifier*. A region identifier is a portable and stable name for a region; in order to access a region, a processor node must know the region's region identifier. New regions can be created dynamically (using `rgn_create`) and can later be removed when they are no longer needed. Blocks of memory comprising distinct regions (those with different region identifiers) are non-overlapping.

In order to access the memory associated with a region, it must be *mapped* into the local address space. To map a region, one calls `rgn_map` with the region identifier of the region to be mapped; the function returns a pointer to the base of the region's data area.

CRL also provides operations for sharing regions. Specifically, CRL provides library calls that can be used to delimit *operations*—portions of the code where regions will be read from or written to. The library calls initiating and terminating operations are required to maintain the consistency of shared data. The coherence model provided by CRL considers entire operations on regions as indivisible units. From this perspective, CRL provides sequential consistency for read and write operations in the same sense that a sequentially-consistent hardware-based DSM does for individual loads and stores.

In order to use CRL functionality, application source files should include the file `crl.h`. When compiling, users of the TCP/UNIX implementation should make sure that the symbol `TCPUNIX` is `#define`-ed before including `crl.h` (*e.g.,* before the `#include` or via a compiler directive) to ensure that slight differences in the CRL interface (see Section 6) necessary for the TCP/UNIX version are reflected in the include file. After compilation, application object files should be linked with the CRL object file (`libcrl.o`) to produce an executable.

3.1 CRL Functions

This section provides a brief description of the functionality provided by CRL 1.0.

3.1.1 Basic CRL Functions and Variables

`void crl_init(void)`
: Initialize the CRL internals; `crl_init` should be called on all nodes simultaneously. `crl_init` should be called before any CRL functionality is utilized. (The TCP/UNIX

version of CRL 1.0 uses a slightly different interface to `crl_init`; see Section 6 for details.)

`unsigned crl_num_nodes`
: Contains the number of nodes in the system. (*This may be removed or altered in future releases.*)

`unsigned crl_self_addr`
: Contains the index of the node on which a thread is running. Each node has a unique index between 0 and `crl_num_nodes`-1; this index can be used to allow programs to perform different tasks on different nodes. (*This may be removed or altered in future releases.*)

`typedef unsigned rid_t`
: `rid_t` is a scalar type that is used to represent region identifiers; region identifiers are portable, unique 'names' for regions. In this implementation, region identifiers are represented using (32-bit) unsigned integers.

3.1.2 Basic Region Functions

`rid_t rgn_create(unsigned size)`
: Create a new region of size `size` bytes (initially resident on the calling node) and return a region identifier for the new region. The data area of newly created regions is not initialized.

`void rgn_destroy(rid_t rgn_id)`
: Destroy the region named by region identifier `rgn_id`. (*In the current implementation,* `rgn_destroy` *does nothing.*)

`rid_t rgn_rid(void *rgn)`
: Return the region identifier of the region represented by `rgn`, where `rgn` is a pointer that was returned by `rgn_map` (see subsection 3.1.3).

`unsigned rgn_size(void *rgn)`
: Return the size (in bytes) of the region represented by `rgn`, where `rgn` is a pointer that was returned by `rgn_map` (see subsection 3.1.3).

`void rgn_flush(void *rgn)`
: Flush the local copy of the region represented by `rgn`, where `rgn` is a pointer that was returned by `rgn_map` (see subsection 3.1.3). When sensible to do so, `rgn_flush` both writes back any local changes to a region's home node and sends a protocol message relinquishing read ownership.

3.1.3 Mapping and Unmapping Region Functions

`void *rgn_map(rid_t rgn_id)`
: Map the region named by `rgn_id` into the address space of the local node. Return a pointer to the base of the region's data area; the pointer is guaranteed to be doubleword aligned. This pointer is also used to name a mapped region when initiating and terminating operations or unmapping the region. A region can be mapped multiple times on the same node.

The address at which a region is mapped into the local address space is only guaranteed to remain fixed for the duration of a mapping; subsequent mappings of the same region may yield different addresses. Further, two mappings of the same region on different nodes are not guaranteed to be at the same address. Because of this, the only globally portable and stable name for a region is the corresponding region identifier. Thus, applications that need to store a "pointer" to shared data (*e.g.,* in another region as part of a distributed, shared data structure) must store the corresponding region's unique identifier (as returned by `rgn_create`), *not* the address at which the region is currently mapped. Subsequent references to the data referenced by the region identifier must be preceded by calls to `rgn_map` (to obtain the address at which the region is mapped) and followed by calls to `rgn_unmap` (to clean up the mapping).

`void rgn_unmap(void *rgn)`
Terminates the mapping represented by `rgn`, where `rgn` is a pointer that was returned by `rgn_map`. If a region is mapped multiple times on the calling node, `rgn_unmap` only terminates one 'reference' to the mapped region.

3.1.4 Region Read and Write Operations

`void rgn_start_read(void *rgn)`
Initiate a read operation on the region represented by `rgn`, where `rgn` is a pointer that was returned by `rgn_map`. The read operation is considered to be in progress from the time `rgn_start_read` returns until a corresponding call to `rgn_end_read`; during this time, the contents of the region's data area may be safely read.

If any write operations are in-progress or pending on a region, `rgn_start_read` will block (not return) until they have been completed. Multiple read operations on the same region may be in progress simultaneously.

`void rgn_end_read(void *rgn)`
Terminate a read operation on the region represented by `rgn`, where `rgn` is a pointer that was returned by `rgn_map`. If multiple read operations are in progress on the calling node, `rgn_end_read` only terminates one of them.

`void rgn_start_write(void *rgn)`
Initiate a write operation on the region represented by `rgn`, where `rgn` is a pointer that was returned by `rgn_map`. The write operation is considered to be in progress from the time `rgn_start_write` returns until a corresponding call to `rgn_end_write`; during this time, the contents of the region's data area may be safely read or written.

If any read or write operations are in-progress or pending on a region, `rgn_start_write` will block (not return) until they have been completed. Because write operations are serialized with respect to all other operations on the same region, no other operations can be in progress during a write operation (including subsequent read or write operations initiated at the node with a write operation already in progress).

`void rgn_end_write(void *rgn)`
Terminate a write operation on the region represented by `rgn`, where `rgn` is a pointer that was returned by `rgn_map`.

3.1.5 Global Synchronization Functions

`void rgn_barrier(void)`
Participate in a global barrier with all other nodes. `rgn_barrier` does not return on any node until it has been called on all nodes.

`void rgn_bcast_send(int len, void *buf)`
Initiate a global broadcast to all other nodes of `len` bytes starting at address `buf`. All other nodes must call `rgn_bcast_recv` with the same `len` argument and a pointer to an appropriately-sized buffer.

`void rgn_bcast_recv(int len, void *buf)`
Receive a global broadcast initiated by another node. The value given for `len` must match that provided on the node initiating the broadcast and `buf` must point at an appropriately-sized buffer.

`double rgn_reduce_dadd(double arg)`
Participate in a global reduction. `rgn_reduce_dadd` does not return on any node until it has been called on all nodes. Returns the sum of the `arg` values provided on each node.

`double rgn_reduce_dmin(double arg)`
Participate in a global reduction. `rgn_reduce_dmin` does not return on any node until it has been called on all nodes. Returns the minimum of the `arg` values provided on each node.

`double rgn_reduce_dmax(double arg)`
Participate in a global reduction. `rgn_reduce_dmax` does not return on any node until it has been called on all nodes. Returns the maximum of the `arg` values provided on each node.

3.1.6 Replacements for Standard Functions

CRL 1.0 provides replacement functions for `malloc` and `free`. This is necessary because CRL protocol message handlers use `malloc` to dynamically allocate memory (when necessary). Since message handlers can interrupt the "normal" computation running on a node and standard implementations of `malloc` and other memory allocation functions are not reentrant, this can lead to serious problems if care isn't taken to ensure that calls to `malloc` and `free` are executed atomically.

`void *safe_malloc(unsigned nbytes)`
When using CRL, calls to `malloc` should be replaced with calls to `safe_malloc`. Calls to other `malloc`-like functions (*e.g.,* `calloc`, `realloc`) should either be replaced with calls to `safe_malloc` or by implementing new `safe_xxx` functions using `safe_malloc` as a template.

`void safe_free(void *ptr)`
When using CRL, calls to `free` should be replaced with calls to `safe_free`.

4 General Caveats

The CM-5 and Alewife implementations of CRL only support 'single-threaded' applications (applications with a single thread/process per processor). The TCP/UNIX version allows a single thread for each PVM process (but multiple PVM processes may reside on the same physical machine).

For the most part, this implementation of CRL fails catastrophically if a problem is encountered instead of doing something reasonable like returning an informative error code.

After calling `crl_init`, applications should use `safe_malloc` and `safe_free` instead of `malloc` and `free` when allocating or deallocating memory. (See Section 3.1.6 for details.)

As distributed, the source code for CRL 1.0 will not work on systems with larger than 256 processors. (See the `#define` for `MaxNodesBits` in `crl_int.h`.)

See the "Platform-Specific Issues" subsections in each of the following sections for other caveats specific to a particular platform.

5 Using the CM-5 Version of CRL

This section contains information relevant to building and using the CM-5 version of CRL.

The CM-5 version of CRL 1.0 was developed and debugged on a 128-node system with Ross (32 MHz SPARC v.7) processors; the system was running CMOST 7.4 Final using the CMMD 3.3 message-passing libraries. We believe that CRL should also work fine on CM-5 systems with new (SuperSPARC "Viking") processors and/or those running the previous software release (CMOST 7.3 and CMMD 3.2).

5.1 Building CRL for the CM-5

To build CRL 1.0 for the CM-5:

1. Obtain a copy of the CRL 1.0 distribution (`crl-1.0.tar.gz`). This can be accomplished via links from the CRL World Wide Web page (`http://www.pdos.lcs.mit.edu/crl/`) or via anonymous ftp from `cag.lcs.mit.edu` in `pub/crl`.
2. Make a copy of `crl-1.0.tar.gz` in the directory where you want to install CRL 1.0; `cd` to that directory.
3. The CRL 1.0 tar file is compressed using `gzip` (thus the `.gz` extension). Uncompress the tar file using `gunzip`:

   ```
   unix% gunzip crl-1.0.tar.gz
   ```

   This will produce an uncompressed version of `crl-1.0.tar.gz` named `crl-1.0.tar` (and will also remove the compressed version).
4. Untar the distribution using `tar xf`:

   ```
   unix% tar xf crl-1.0.tar
   ```

This will create a subdirectory of the current working directory called `crl-1.0` and unpack the CRL 1.0 distribution into it.

5. `cd` into the CRL 1.0 `src` directory:

```
unix% cd crl-1.0/src
```

   This directory contains the sources for the CRL library.

6. Use the `make` utility and the supplied `Makefile.CM5` file to build the CRL 1.0 object file (`libcrl.o`) for the CM-5 version of CRL:

```
unix% make -f Makefile.CM5
```

   Once this completes, you are done building CRL. Applications intended for use with the CM-5 version of CRL should be linked against the resulting object file (`libcrl.o`).

To build and run the example application shown in Appendix A:

1. `cd` to the `apps/example` subdirectory of the top-level CRL 1.0 directory (`crl-1.0`). If you have just finished following the directions given above for building CRL, this would be accomplished by something like:

```
unix% cd ../apps/example
```

2. Use the `make` utility and the supplied `Makefile.CM5` file to compile the application and link it against the necessary object files (*e.g.,* `libcrl.o`) and libraries.

```
unix% make -f Makefile.CM5
```

   The end result of this process should be a ready-to-run `example` binary.

3. Assuming DJM is being used on your CM-5 for running jobs, the example application could then be run using something like:

```
unix% jrun -proc 32 -mem 100m -time 1min -- example
```

Applications in the other subdirectories of `crl-1.0/apps` can be compiled and run in a similar manner.

5.2 Platform-Specific Issues

When using CRL on the CM-5, the programmer is allowed to disable interrupts (message delivery). If this is done, the user is responsible for explicitly polling the network and should not assume anything about whether or not calling CRL functions will cause any messages to be drained from the network.

The point where the performance of the two bulk transfer mechanisms (`scopy` and that implemented in `cm5_fast_data.c`) crosses over (and thus CRL decides to switch between them) is probably very platform (hardware, CMOST, and CMMD version) dependent. The value supplied in the CRL

1.0 sources (256 bytes) was obtained through hand tuning on our development platform. (See the comment above the `#define` for `DataXover` in `proto.c`.)

The Makefiles supplied for use with the CM-5 version of CRL 1.0 assume that the GNU C compiler (`gcc`) is available. If `gcc` is not available in your local environment or you would like to use a different C compiler, you will need to edit the `Makefile.CM5` files in `crl-1.0/src` and `crl-1.0/apps/*` and change the `CC = gcc` lines to name the C compiler that should be used instead.

6 Using the TCP/UNIX Version of CRL

This section contains information relevant to building and using the TCP/UNIX version of CRL.

The TCP/UNIX version of CRL 1.0 was developed and debugged on a network of uniprocessor Sun SPARCstations (of various kinds) running SunOS 4.1.3. Interprocessor communication is effected over standard 10 Mbit/second ethernet using TCP sockets. We expect that the core elements of the TCP/UNIX version of CRL will work on other UNIX-like hardware/software platforms with only modest porting effort.

In order to use the TCP/UNIX version of CRL, PVM [5] must be installed on the system; this allows CRL to leverage off of the group and process management facilities provided by PVM instead of reimplementing them from scratch. Interestingly, this also allows applications running under the TCP/UNIX version of CRL to freely intermix invocations of CRL and PVM functionality (with one exception; see the description of `crl_init` in Section 6.2 below). We have tested CRL 1.0 against versions 3.3.7 and 3.3.8 of PVM.

The TCP/UNIX implementation included in CRL 1.0 is not intended to be a high-performance distributed shared memory implementation delivering speedups similar to those obtained with the Alewife and the CM-5 implementations. It is provided to allow development and experimentation by those without access to a CM-5 and to display the ease with which CRL may be ported.

6.1 Building CRL for TCP/UNIX

To build the TCP/UNIX version of CRL 1.0:

1. Obtain a copy of the CRL 1.0 distribution (`crl-1.0.tar.gz`). This can be accomplished via links from the CRL World Wide Web page (`http://www.pdos.lcs.mit.edu/crl/`) or via anonymous ftp from `cag.lcs.mit.edu` in `pub/crl`.

2. Make a copy of `crl-1.0.tar.gz` in the directory where you want to install CRL 1.0; `cd` to that directory.

3. The CRL 1.0 tar file is compressed using `gzip` (thus the `.gz` extension). Uncompress the tar file using `gunzip`:

   ```
   unix% gunzip crl-1.0.tar.gz
   ```

   This will produce an uncompressed version of `crl-1.0.tar.gz` named `crl-1.0.tar` (and will also remove the compressed version).

4. Untar the distribution using `tar xf`:

   ```
   unix% tar xf crl-1.0.tar
   ```

   This will create a subdirectory of the current working directory called `crl-1.0` and unpack the CRL 1.0 distribution into it.

5. `cd` into the CRL 1.0 `src` directory:

   ```
   unix% cd crl-1.0/src
   ```

   This directory contains the sources for the CRL library.

6. Edit `Makefile.TCPUNIX` and set the `PVM_ROOT` variable to point to the PVM home directory.

7. Use the `make` utility and the supplied `Makefile.TCPUNIX` file to build the CRL 1.0 object file (`libcrl.o`) for the TCP/UNIX version of CRL:

   ```
   unix% make -f Makefile.TCPUNIX
   ```

   Once this completes, you are done building CRL. Applications intended for use with the TCP/UNIX version of CRL should be linked against the resulting object file (`libcrl.o`).

To build and run the example application shown in Appendix A:

1. `cd` to the `apps/example` subdirectory of the top-level CRL 1.0 directory (`crl-1.0`). If you have just finished following the directions given above for building CRL, this would be accomplished by something like:

   ```
   unix% cd ../apps/example
   ```

2. Use the `make` utility and the supplied `Makefile.TCPUNIX` file to compile the application and link it against the necessary object files (*e.g.,* `libcrl.o`) and libraries (*e.g.,* those needed for PVM). You may need to edit `Makefile.TCPUNIX` and change the `PVM_ROOT` and `PVM_ARCH` variables.

   ```
   unix% make -f Makefile.TCPUNIX
   ```

3. If the PVM daemons are not running or the PVM virtual machine is not setup yet, start the PVM daemon on the local machine with:

   ```
   unix% pvm
   ```

   This should put you in the PVM console application. The `help` command will give a short description of each console command. The most useful for us are

   - conf - List virtual machine configuration
   - add <hostname> - Add hosts to virtual machine
   - delete <hostname> - Delete hosts from virtual machine
   - ps ax - List all tasks on all hosts in the virtual machine

- reset - Kill all tasks
- quit - Exit console
- halt - Stop pvm daemons on every host in the virtual machine

Use the `add` command to create the virtual machine. There are two common problems when using `add` to add hosts:

- When starting a non-interactive shell, PVM requires that no extraneous output (text) be generated. If there is some output, then put the line

  ```
  if (! $?prompt) exit
  ```

  in your `.cshrc` before anything is output. This line causes non-interactive shells to stop executing `.cshrc`.
- You must be able to `rsh` to the remote host you're trying to add. If you are having trouble with this, consult your local system adminstrator (it may require things like setting up an `.rhosts` file or acquiring appropriate Kerberos tickets).

For more information on starting up PVM, please see the online book on the WWW at `http://www.netlib.org/pvm3/book/pvm-book.html`. The web page also contains instructions for obtaining a postscript version of the book (which is more complete than the online version).

4. In order for PVM to spawn processes successfully, it is necessary to add the following to your `.cshrc` file:

   ```
   setenv PVM_ROOT /home/crl/pvm3
   setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`
   set path=($path $PVM_ROOT/lib/$PVM_ARCH \
            ~/pvm3/bin/$PVM_ARCH $PVM_ROOT/bin/$PVM_ARCH)
   ```

   The first `setenv` should set `PVM_ROOT` to wherever PVM is installed on your local system.

5. If the `example` binary can be found in your `$path` (*e.g.,* by adding a symbolic link to it from `~/pvm3/bin/$PVM_ARCH` and running `rehash`), then the sample applications can be started simply by executing the application. Because of the way that `pvm_app.c` is written, no relative pathnames can be used when invoking the executable (*e.g.,* `./example` won't work); only executable names or absolute pathnames are acceptable.

   ```
   unix% example
   ```

Applications in the other subdirectories of `crl-1.0/apps` can be compiled and run in a similar manner.

6.2 Platform-Specific Issues

In order to enable leveraging off of PVM's group and process management functionality, the interface to `crl_init` is slightly different under the TCP/UNIX version of CRL:

`void crl_init(char *groupname)`

`groupname` is the name of the PVM process group upon which CRL should be initialized; `crl_init` should be called by all member processes simultaneously. `crl_init` should be called before any CRL functionality is utilized.

No node should call `crl_init` before all the other nodes have joined the PVM group. After CRL has been initialized, no process is allowed to leave the group (nor are new processes allowed to join it). Also, the instance number assigned to each process must be consecutively assigned from 0 to `crl_num_nodes`-1. This could only be a problem if processes had previously left the group.

In the TCP/UNIX version of CRL, `crl_init` will block until network connections have been established with all other processes in the PVM group. If insufficient network resources are available to establish these connections, CRL will probably die catastrophically instead of doing something useful (*e.g.,* returning an error code or printing a useful error message).

In order to ensure graceful and coordinated termination of a group of PVM processes running CRL, the TCP/UNIX implementation also provides a `crl_exit` function to ensure that all nodes exit cleanly and help eliminate mysterious errors.

`void crl_exit(void)`

Clean up after CRL and coordinate a graceful termination with other PVM processes in the same CRL group. `crl_exit` should be called before a process exists (but after it has finished using all CRL functionality).

The sample `main()` in `apps/pvm.common/pvm_app.c` is an example of how to start an application using the TCP/UNIX version of CRL. All of the applications included in the `apps` directory use this sample `main()`. See the README file in `apps/pvm.common` for information on what `main()` is doing. In general, an application using the TCP/UNIX implementation of CRL must do the following during startup:

1. Enroll the application with PVM: `pvm_mytid()`
2. Join a PVM group: `pvm_joingroup(GROUP)`
3. Wait for other processes to join the PVM group: `pvm_barrier(GROUP, numnodes)`
4. Initialize CRL: `crl_init()`

The applications (or symbolic links to the applications) spawned by the PVM process management code must be available in the path searched by PVM.

Even though an application may be started from an arbitrary place in the filesystem, each of the processes spawned by PVM initially executes in the user's home directory. Unless the working directory of the spawned application is changed (using standard C library calls), any files accessed by the application must have pathnames relative to the home directory.

The Makefiles supplied for use with the TCP/UNIX version of CRL 1.0 assume that the GNU C compiler (`gcc`) is available. If `gcc` is not available in your local environment or you would like to use a different C compiler, you will need to edit the `Makefile.TCPUNIX` files in `crl-1.0/src`

and `crl-1.0/apps/*` and change the `CC = gcc` lines to name the C compiler that should be used instead.

The TCP/UNIX version of CRL 1.0 uses `SIGIO` to implement active-message-like functionality using TCP sockets. Mixing CRL with source code that manipulate signals for other reasons should only be done with the greatest of care.

6.3 Obtaining PVM

Further information about PVM (including sources) can be obtained via the PVM World Wide Web Home Page (`http://www.epm.ornl.gov/pvm/`).

7 Using the Null Version of CRL

This section contains information relevant to building and using the null version of CRL.

The null CRL implementation is provided to allow sequential timings of CRL applications to be obtained without any overhead due to CRL. It can be used both on a single processor of a parallel system (*e.g.,* CM-5, Alewife) on on uniprocessor systems (*e.g.,* desktop workstations).

7.1 Building the Null Version of CRL

To build the null version of CRL 1.0:

1. Obtain a copy of the CRL 1.0 distribution (`crl-1.0.tar.gz`). This can be accomplished via links from the CRL World Wide Web page (`http://www.pdos.lcs.mit.edu/crl/`) or via anonymous ftp from `cag.lcs.mit.edu` in `pub/crl`.

2. Make a copy of `crl-1.0.tar.gz` in the directory where you want to install CRL 1.0; `cd` to that directory.

3. The CRL 1.0 tar file is compressed using `gzip` (thus the `.gz` extension). Uncompress the tar file using `gunzip`:

   ```
   unix% gunzip crl-1.0.tar.gz
   ```

   This will produce an uncompressed version of `crl-1.0.tar.gz` named `crl-1.0.tar` (and will also remove the compressed version).

4. Untar the distribution using `tar xf`:

   ```
   unix% tar xf crl-1.0.tar
   ```

   This will create a subdirectory of the current working directory called `crl-1.0` and unpack the CRL 1.0 distribution into it.

5. `cd` into the CRL 1.0 `null` directory:

```
unix% cd crl-1.0/null
```

This directory contains the sources for the null version of the CRL library.

6. Use the `make` utility and the supplied `Makefile.null` file to build the CRL 1.0 object file (`libcrl.o`) for the null version of CRL:

```
unix% make -f Makefile.null
```

Once this completes, you are done building CRL. Applications intended for use with the null version of CRL should be linked against the resulting object file (`libcrl.o`).

To build and run the example application shown in Appendix A on a desktop workstation:

1. `cd` to the `apps/example` subdirectory of the top-level CRL 1.0 directory (`crl-1.0`). If you have just finished following the directions given above for building CRL, this would be accomplished by something like:

```
unix% cd ../apps/example
```

2. Use the `make` utility and the supplied `Makefile.null` file to compile the application and link it against the necessary object files (*e.g.,* `libcrl.o`).

```
unix% make -f Makefile.null
```

3. Run the `example` application with no arguments

```
unix% example
```

Applications in the other subdirectories of `crl-1.0/apps` can be compiled and run in a similar manner.

7.2 Platform-Specific Issues

The directions given above are known to work on Sun SPARCstations running SunOS 4.1.3; we expect that little or no change would be necessary for use with other hardware/software combinations.

8 Obtaining CRL 1.0

The CRL 1.0 distribution can be obtained via links from the CRL World Wide Web page (URL `http://www.pdos.lcs.mit.edu/crl/`). It can also be obtained via anonymous ftp from `cag.lcs.mit.edu`; cd to `/pub/crl` and get the file `crl-1.0.tar.gz`. The distribution contains sources both for CRL 1.0 and for several sample applications.

9 Copyright Notice

The bulk of the CRL 1.0 distribution (both source code and supporting documentation) is covered by the following copyright:

Copyright (C) 1995 Massachusetts Institute of Technology

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. The author makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

THE AUTHORS DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Some parts of the CRL 1.0 distribution (as marked) are covered by other copyright holders/polices (*e.g.,* Water and Barnes-Hut code derived from the Splash-2 distribution).

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[3] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[4] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

[5] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4), December 1990.

[6] Thinking Machines Corporation. *CMMD Reference Manual, Version 3.0*, May 1993.

A Example

This section contains source code for a well-documented example of a simple application that uses CRL. The same source code and supporting Makefiles can be found in `apps/example`.

```
/*
 * example.c
 * joseph adler
 * august 1995
 *
 * Copyright (C) 1995 Massachusetts Institute of Technology
 *
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without
 * fee, provided that the above copyright notice appear in all copies
 * and that both that copyright notice and this permission notice
 * appear in supporting documentation. The author makes no
 * representations about the suitability of this software for any
 * purpose. It is provided "as is" without express or implied
 * warranty.
 *
 * THE AUTHORS DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
 * NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
 * OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
 * NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
 * CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *
 * RCS Id: example.c,v 1.12 1995/08/24 08:24:34 tuna Exp
 */

/* This example program is designed to help guide the user through
 * programming a simple program with CRL. It is a simple matrix
 * multiply program, with some extra hooks for dividing work in some
 * reasonable fashion.
 *
 * Simple serial version, adopted from:
 *
 *  _Introduction to Algorithms_. Thomas H. Cormen, Charles
 *  E. Leiserson, and Ronald L. Rivest. MIT Press, 1990.
 *
 *  void mm(int a[N][N], int b[N][N], int c[N][N])
 *  {
 *    int i, j, k;
 *
 *    for (i=0; i<N; i++)
 *    {
 *      for (j=0; j<N; j++)
 *      {
 *        c[i][j] = 0;
 *        for (k=0; k<N; k++)
 *          c[i][j] = c[i][j] + (a[i][k] * b[k][j]);
```

```
 *      }
 *    }
 *  }
 *
 * Because the purpose of this example is to illustrate how to program
 * in CRL, the example is intentionally kept quite simple (e.g., the
 * two input matrices are hard coded; the particular work and data
 * decomposition used is not particularly efficient).
 */

/* We begin by #including "crl.h", #defining some constants, and
 * declaring the two input matrices.
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#if defined(CM5)
#include <cm/cmmd.h>
#elif defined(ALEWIFE)
#include <parallel.h>
#endif

#include "crl.h"

/* MASTER is the node responsible for creating the regions
 * used to represent the destination matrix (c)
 */
#define MASTER (0)

/* N is the size of the (square) matrices used
 */
#define N (5)

/* IntMatrix is a typedef for an N by N matrix of int elements
 */
typedef int IntMatrix[N][N];

/* RidMatrix is a typedef for an N by N matrix of rid_t elements
 * (an rid_t is a CRL region identifier)
 */
typedef rid_t RidMatrix[N][N];

/* the first input matrix (static; replicated on all processors)
 */
IntMatrix a = {{ 2,  0,  0,  0,  0},
               { 0,  2,  0,  0,  0},
               { 0,  0,  2,  0,  0},
               { 0,  0,  0,  2,  0},
               { 0,  0,  0,  0,  2}};

/* the second input matrix (static; replicated on all processors)
 */
```

```
IntMatrix b = {{ 1,  2,  3,  4,  5},
               { 6,  7,  8,  9, 10},
               {11, 12, 13, 14, 15},
               {16, 17, 18, 19, 20},
               {21, 22, 23, 24, 25}};

/* Now we need to consider the question of how we divide the work
 * involved in matrix multiplication. Basically, we want the program
 * to proceed as follows:
 *
 * 1. The MASTER node creates the regions used to represent the
 *    destination matrix (c).
 *
 * 2. Each node does its share of the computation and stops at a
 *    barrier when done.
 *
 * 3. After nodes return from the barrier, all work is done. The
 *    MASTER node prints out the values found in the destination
 *    matrix and all nodes exit.
 *
 * To implement this, we create a matrix of region identifiers, each
 * of which will in turn refer to a region which contains the value
 * of that element of the destination matrix (c).
 */
RidMatrix c;

/* We divide work in this program so that each node is called upon to
 * perform approximately the same amount of work. The scheme which is
 * used is fairly simple: we color each square in the matrix with a
 * node index number, traversing the n by n matrix from left to right,
 * and top to bottom, counting off node indices in ascending order. For
 * example, with seven nodes and a 4 by 4 matrix, assignments would
 * look like:
 *
 *  0  1  2  3
 *  4  5  6  0
 *  1  2  3  4
 *  5  6  0  1
 */

/* If we're using the TCP/UNIX version of CRL, the main procedure can
 * be found in ../pvm.common/pvm_app.c. This procedure takes care of
 * setting up a pvm group, and spawning processes. It then calls
 * main2(). Please see this code for more information on how to set up
 * a program to run using pvm.
 *
 * If we're running CRL on any other platform, then we need to include
 * a main procedure that calls main2().
 */

extern int main2(int, char **);

#if defined(TCPUNIX)
extern char *GROUP;
```

```
#else
int main(int argc, char **argv)
{
#if defined(CM5)
  CMMD_set_io_mode(0, CMMD_independent);
  CMMD_set_io_mode(1, CMMD_independent);
  CMMD_set_io_mode(2, CMMD_independent);
#endif

#if defined(ALEWIFE)
  do_in_parallel(main2, argc, argv);
#else
  main2(argc, argv);
#endif

  return 0;
}
#endif

int main2(int argc, char **argv)
{
  int  index;
  int  row, col;
  int  k;
  int *tmp;

  /* initialize CRL
   */
#if defined(TCPUNIX)
  crl_init(GROUP);
#else
  crl_init();
#endif

  if (crl_self_addr == MASTER)
  {
    /* on the MASTER node, create all the regions used to represent
     * the destination matrix
     */
    for (row=0; row<N; row++)
      for (col=0; col<N; col++)
        c[row][col] = rgn_create(sizeof(int));

    /* broadcast this set of region identifiers to all other nodes
     */
    rgn_bcast_send(sizeof(RidMatrix), c);
  }
  else
  {
    /* receive the set of region identifiers broadcast by the MASTER
     * node
     */
    rgn_bcast_recv(sizeof(RidMatrix), c);
  }
```

```
/* work distribution algorithm:
 *
 * Each node then computes the value of all destination matrix
 * elements where the index of the element in the matrix (numbering
 * from left to right, top to bottom) modulo the number of nodes is
 * equal to the local node index.
 *
 * that is, if
 *
 *  index = (N * row) + col
 *
 * each node computes all destination matrix elements for which
 *
 *  (index % crl_num_nodes) == crl_self_addr
 */

for (index=crl_self_addr; index<(N*N); index+=crl_num_nodes)
{
  row = index / N;
  col = index % N;

  printf("node %d computing c[%d][%d]\n", crl_self_addr, row, col);

  /* now, map the region representing the current destination matrix
   * element and initiate a write so we can write to it
   */
  tmp = rgn_map(c[row][col]);
  rgn_start_write(tmp);

  /* compute the value of this element of the destination matrix
   */
  *tmp = 0;
  for (k=0; k<N; k++)
    *tmp += (a[row][k] * b[k][col]);

  /* since we're done with this element of the destination matrix,
   * terminate the write operation and unmap the region
   */
  rgn_end_write(tmp);
  rgn_unmap(tmp);
}

/* wait for all nodes to finish computing
 */
rgn_barrier();

/* MASTER node prints out the destination matrix
 */
if (crl_self_addr == MASTER)
{
  printf("Final matrix:\n");
  for (row=0; row<N; row++)
  {
```

```
      for (col=0; col<N; col++)
      {
        /* map the region representing the current element and
         * initiate a read operation
         */
        tmp = (int *) rgn_map(c[row][col]);
        rgn_start_read(tmp);

        printf("%6d ", *tmp);

        /* terminate the read operation and unmap the region
         */
        rgn_end_read(tmp);
        rgn_unmap(tmp);
      }

      printf("\n");
    }
  }

#if defined(TCPUNIX)
  crl_exit();
#endif
}
```