Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Charm++
Debugger Manual

Version 0.2

University of Illinois
Charm++/Converse Parallel Programming System Software
Non-Exclusive, Non-Commercial Use License

Upon execution of this Agreement by the party identified below ("Licensee"), The Board of Trustees of the University of Illinois ("Illinois"), on behalf of The Parallel Programming Laboratory ("PPL") in the Department of Computer Science, will provide the Charm++/Converse Parallel Programming System software ("Charm++") in Binary Code and/or Source Code form ("Software") to Licensee, subject to the following terms and conditions. For purposes of this Agreement, Binary Code is the compiled code, which is ready to run on Licensee's computer. Source code consists of a set of files which contain the actual program commands that are compiled to form the Binary Code.

1. The Software is intellectual property owned by Illinois, and all right, title and interest, including copyright, remain with Illinois. Illinois grants, and Licensee hereby accepts, a restricted, non-exclusive, non-transferable license to use the Software for academic, research and internal business purposes only, e.g. not for commercial use (see Clause 7 below), without a fee.

2. Licensee may, at its own expense, create and freely distribute complimentary works that interoperate with the Software, directing others to the PPL server (`http://charm.cs.illinois.edu`) to license and obtain the Software itself. Licensee may, at its own expense, modify the Software to make derivative works. Except as explicitly provided below, this License shall apply to any derivative work as it does to the original Software distributed by Illinois. Any derivative work should be clearly marked and renamed to notify users that it is a modified version and not the original Software distributed by Illinois. Licensee agrees to reproduce the copyright notice and other proprietary markings on any derivative work and to include in the documentation of such work the acknowledgement:

   > "This software includes code developed by the Parallel Programming Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign."

   Licensee may redistribute without restriction works with up to 1/2 of their non-comment source code derived from at most 1/10 of the non-comment source code developed by Illinois and contained in the Software, provided that the above directions for notice and acknowledgement are observed. Any other distribution of the Software or any derivative work requires a separate license with Illinois. Licensee may contact Illinois (`kale@illinois.edu`) to negotiate an appropriate license for such distribution.

3. Except as expressly set forth in this Agreement, THIS SOFTWARE IS PROVIDED "AS IS" AND ILLINOIS MAKES NO REPRESENTATIONS AND EXTENDS NO WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY PATENT, TRADEMARK, OR OTHER RIGHTS. LICENSEE ASSUMES THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS. LICENSEE AGREES THAT UNIVERSITY SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES WITH RESPECT TO ANY CLAIM BY LICENSEE OR ANY THIRD PARTY ON ACCOUNT OF OR ARISING FROM THIS AGREEMENT OR USE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS.

4. Licensee understands the Software is proprietary to Illinois. Licensee agrees to take all reasonable steps to insure that the Software is protected and secured from unauthorized disclosure, use, or release and will treat it with at least the same level of care as Licensee would use to protect and secure its own proprietary computer programs and/or information, but using no less than a reasonable standard of care. Licensee agrees to provide the Software only to any other person or entity who has registered with Illinois. If licensee is not registering as an individual but as an institution or corporation each member of the institution or corporation who has access to or uses Software must agree to and abide by the terms of this license. If Licensee becomes aware of any unauthorized licensing, copying or use of the Software, Licensee shall promptly notify Illinois in writing. Licensee expressly agrees to use the Software only in the manner and for the specific uses authorized in this Agreement.

5. By using or copying this Software, Licensee agrees to abide by the copyright law and all other applicable laws of the U.S. including, but not limited to, export control laws and the terms of this license. Illinois shall have the right to terminate this license immediately by written notice upon Licensee's breach of, or non-compliance with, any terms of the license. Licensee may be held legally responsible for any copyright infringement that is caused or encouraged by its failure to abide by the terms of this license. Upon termination, Licensee agrees to destroy all copies of the Software in its possession and to verify such destruction in writing.

6. The user agrees that any reports or published results obtained with the Software will acknowledge its use by the appropriate citation as follows:

   > "Charm++/Converse was developed by the Parallel Programming Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign."

   Any published work which utilizes Charm++ shall include the following reference:

   > "L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In 'Parallel Programming using C++' (Eds. Gregory V. Wilson and Paul Lu), pp 175-213, MIT Press, 1996."

   Any published work which utilizes Converse shall include the following reference:

   > "L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. Proceedings of the 10th International Parallel Processing Symposium, pp 212-217, April 1996."

   Electronic documents will include a direct link to the official Charm++ page at `http://charm.cs.illinois.edu/`

7. Commercial use of the Software, or derivative works based thereon, REQUIRES A COMMERCIAL LICENSE. Should Licensee wish to make commercial use of the Software, Licensee will contact Illinois (kale@illinois.edu) to negotiate an appropriate license for such use. Commercial use includes:

   (a) integration of all or part of the Software into a product for sale, lease or license by or on behalf of Licensee to third parties, or

   (b) distribution of the Software to third parties that need it to commercialize product sold or licensed by or on behalf of Licensee.

8. Government Rights. Because substantial governmental funds have been used in the development of Charm++/Converse, any possession, use or sublicense of the Software by or to the United States government shall be subject to such required restrictions.

9. Charm++/Converse is being distributed as a research and teaching tool and as such, PPL encourages contributions from users of the code that might, at Illinois' sole discretion, be used or incorporated to make the basic operating framework of the Software a more stable, flexible, and/or useful product. Licensees who contribute their code to become an internal portion of the Software agree that such code may be distributed by Illinois under the terms of this License and may be required to sign an "Agreement Regarding Contributory Code for Charm++/Converse Software" before Illinois can accept it (contact `kale@illinois.edu` for a copy).

UNDERSTOOD AND AGREED.
Contact Information:
The best contact path for licensing issues is by e-mail to `kale@illinois.edu` or send correspondence to:

Prof. L. V. Kale
Dept. of Computer Science
University of Illinois
201 N. Goodwin Ave
Urbana, Illinois 61801 USA
FAX: (217) 244-6500

Contents

1 Introduction

The primary goal of the parallel debugger is to provide an integrated debugging environment which allows the programmer to examine the changing state of the parallel program during the course of its execution.

The Charm++ debugging system has a number of useful features for Charm++ programmers. The system includes a Java GUI client program which runs on the programmer's desktop, and a Charm++ parallel program which acts as a server. The client and server need not be on the same machine, and communicate over the network using a secure protocol described in http://charm.cs.uiuc.edu/manuals/html/converse/5_-CONVERSE_Client_Server_In.html

The system provides the following features:

- Provides a means to easily access and view the major programmer visible entities, including array elements and messages in queues, across the parallel machine during program execution. Objects and messages are extracted as raw data, and interpreted by the debugger, as explained in **??**.
- Provides an interface to set and remove breakpoints on remote entry points, which capture the major programmer-visible control flows in a Charm++ program.
- Provides the ability to freeze and unfreeze the execution of selected processors of the parallel program, which allows a consistent snapshot by preventing things from changing as they are examined.
- Provides a way to attach a sequential debugger to a specific subset of processes of the parallel program during execution, which keeps a manageable number of sequential debugger windows open. Currently these windows are opened independently of the GUI interface, while in the future they will be transformed into an integrated view.

The debugging client provides these features via extensive support built into the Charm++ runtime.

2 Building the Charm++ Debug Tool

To get the Charm Debug tool, check out the source code from the repository. This will create a directory named ccs_tools. Move to this directory and build Charm Debug.

```
git clone https://charm.cs.illinois.edu/gerrit/ccs_tools
cd ccs_tools
ant
```

This will create the executable `bin/charmdebug`, which should work.

You can also download the binaries from the Charm++ downloads website and use it directly without building. (NOTE: Binaries may not be properly working in some platforms, so building from the source code is recommended.)

3 Preparing the Charm++ Application for Debugging

Build Charm++ using `--enable-charmdebug` option. For example:

```
./build charm++ netlrts-darwin-x86_64 --enable-charmdebug
```

No instrumentation is required to use the Charm++ debugger. Being CCS based, you can use it to set and step through entry point breakpoints and examine Charm++ structures on any Charm++ application.

Nevertheless, for some features to be present some additional options might be required at either compile or link time:

- In order to provide a symbol conversion of the assembly code executed by the application, the *-g* option is needed at compile time. This conversion is needed to provide function names as well as source file names and line numbers wherever useful. This is useful also to fully utilize gdb (or any other serial debugger) on one or more processes.

- Optimization options, with their nature of transforming the source code, can produce a mismatch between the function displayed in the debugger (for example in a stack trace) and the functions present in the source code. To produce information coherent with source code, optimization is discouraged.
- The link time option *-memory charmdebug* is only needed if you want to use either the Memory view (see 4.5.1) or the Inspector framework (see 4.5.2) of Charm Debug.

3.1 Record Replay

The *Record Replay* feature is independent of the charmdebug application. It is a mechanism used to detect bugs that happen only once in a while depending on the order in which messages are processed. The program in consideration is first run in a record mode which produces a trace. When the program is run in replay mode it uses a previous trace gotten from a record run to ensure that messages are processed in the same order as the recorded run. The idea is to make use of a message-sequence number and a theorem says that the serial numbers will be the same if the messages are processed in the same order. [**?**]

Record Replay tracing is automatically enabled for Charm++ programs and requires nothing special to be done during compilation (linking with the option "*-tracemode recordreplay*" used to be necessary). At run time, the "*+record*" option is used, which records messages in order in a file for each processor. The same execution order can be replayed using the "*+replay*" runtime option, which can be used at the same time as the other debugging tools in Charm++.

Note! If your Charm++ is built with `CMK_OPTIMIZE` on, all tracing will be disabled. So, use an unoptimized Charm++ to do your debugging.

4 Running the Debugger

4.1 Charmdebug command line parameters

`-pes` Number of PEs

`+p` Number of PEs

`-host` hostname of CCS server for application

`-user` the username to use to connect to the hostname selected

`-port` portnumber of CCS server for application

`-sshtunnel` force the communication between client and server (in particular the one for CCS) to be tunnelled through ssh. This allow the bypass of firewalls.

`-display` X Display

4.2 Basic usage

To run an application locally via the debugger on 4 pes with command line options for your pgm (e.g. opt1 opt2):

```
charmdebug pgm +p4 4 opt1 opt2
```

If the application should be run in a remote cluster behind a firewall, the previous command line will become:

```
charmdebug -host cluster.inst.edu -user myname -sshtunnel pgm +p4 4 opt1 opt2
```

Charmdebug can also be executed without any parameters. The user can then choose the application to launch and its command line parameters from within the `File` menu as shown in Figure 1.

Note: charmdebug command line launching only works on netlrts- and verbs-* builds of Charm++*

To replay a previously recorded session:

```
charmdebug pgm +p4 opt1 opt2  +replay
```

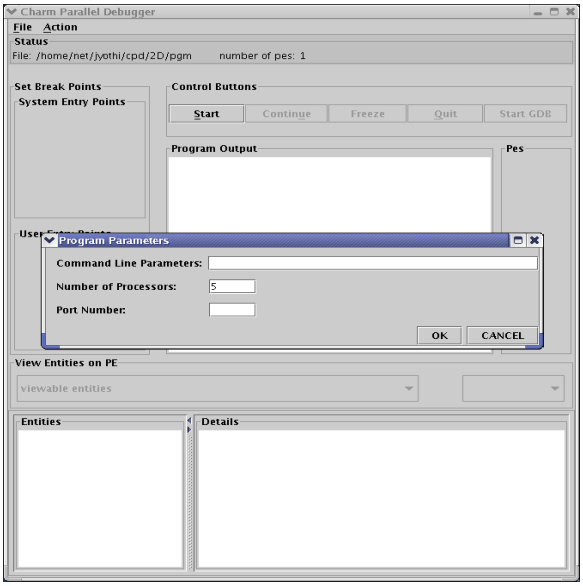

Figure 1: Using the menu to set parameters for the Charm++ program

4.3 Charm Debugging Related Options

When using the charm debugger to launch your application, it will automatically set these to defaults appropriate for most situations.

`+cpd` Triggers application freeze at startup for debugger.

`++charmdebug` Triggers charmrun to provide some information about the executable, as well as provide an interface to gdb for querying.

`+debugger` Which debuggers to use.

`++debug` Run each node under gdb in an xterm window, prompting the user to begin execution.

`++debug-no-pause` Run each node under gdb in an xterm window immediately (i.e. without prompting the user to begin execution).

Note: If you're using the charm debugger it will probably be best to control the sequential (i.e. gdb) debuggers from within its GUI interface.

`++DebugDisplay` X Display for xterm

`++server-port` Port to listen for CCS requests

`++server` Enable client-server (CCS) mode

`+record` Use the recordreplay tracemode to record the exact event/message sequence for later use.

`+replay` Force the use of recorded log of events/messages to exactly reproduce a previous run.

The preceding pair of commands `+record +replay` are used to produce the "instant replay" feature. This feature is valuable for catching errors which only occur sporadically. Such bugs which arise from the nondeterminacy of parallel execution can be fiendishly difficult to replicate in a debugging environment. Typical usage is to keep running the application with +record until the bug occurs. Then run the application under the debugger with the +replay option.

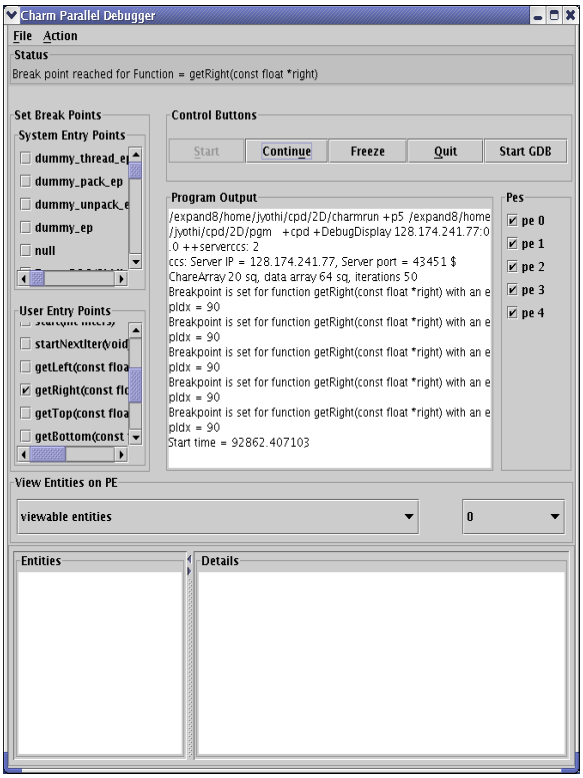

Figure 2: Parallel debugger when a break point is reached

4.4 Charmdebug limitations

4.4.1 Clusters

Charmdebug is currently limited to applications started directly by the debugger due to implementation peculiarities. It will be extended to support connection to remote running applications in the near future.

Due to the current implementation, the debugging tool is limited to netlrts-* and verbs-* versions. Other builds of Charm++ might have unexpected behavior. In the near future this will be extended at least to the mpi-* versions.

4.4.2 Record Replay

The `record replay` feature does not work well with spontaneous events. Load balancing is the most common form of spontaneous event in that it occurs periodically with no other causal event. As per

As per Rashmi's thesis: *There are some unique issues for replay in the context of Charm because it provides high-level support for dynamic load balancing, quiescence detection and information sharing. Many of the load balancing strategies in Charm have a spontaneous component. The strategy periodically checks the sizes of the queues on the local processor. A replay load balancing strategy implements the known load redistribution. The behavior of the old balancing strategy is therefore not replayed only its effect is. Since minimal tracing is used by the replay mechanism the amount of perturbation due to tracing is reduced. The replay mechanism is proposed as a debugging support to replay asynchronous message arrival orders.*

Moreover, if your application crashes without a clean shutdown, the log may be lost with the application.

4.5 Using the Debugger

Once the debugger's GUI loads, the programmer triggers the program execution by clicking the *Start* button. When starting by command line, the application is automatically started. The program starts off displaying

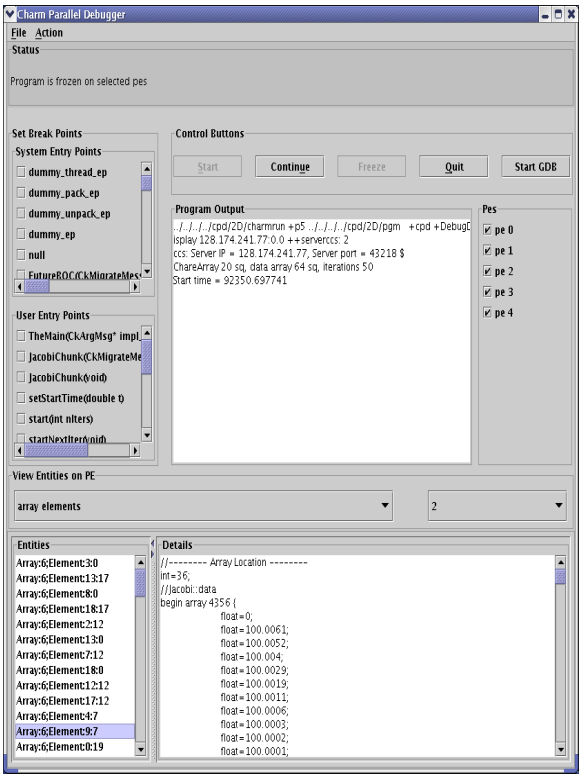

Figure 3: Freezing program execution and viewing the contents of an array element using the Parallel Debugger

the user and system entry points as a list of check boxes, freezing at the onset. The user could choose to set breakpoints by clicking on the corresponding entry points and kick off execution by clicking the *Continue* Button. Figure 2 shows a snapshot of the debugger when a breakpoint is reached. The program freezes when a breakpoint is reached.

Clicking the *Freeze* button during the execution of the program freezes execution, while *Continue* button resumes execution. *Quit* button can be used to abort execution at any point of time. Entities (for instance, array elements) and their contents on any processor can be viewed at any point in time during execution as illustrated in Figure 3.

Specific individual processes of the Charm++ program can be attached to instances of *gdb* as shown in Figure 4. The programmer chooses which PEs to connect *gdb* processes to via the checkboxes on the right side. *Note!* While the program is suspended in gdb for step debugging, the high-level features such as object inspection will not work.

Charm++ objects can be examined via the *View Entities on PE : Display* selector. It allows the user to choose from *Charm Objects, Array Elements, Messages in Queue, Readonly Variables, Readonly Messages, Entry Points, Chare Types, Message Types and Mainchares*. The right sideselector sets the PE upon which the request for display will be made. The user may then click on the *Entity* to see the details.

4.5.1 Memory View

The menu option Action → Memory allows the user to display the entire memory layout of a specific processor. An example is shown in figure 5. This layout is colored and the colors have the following meaning:

red memory allocated by the Charm++ Runtime System;

blue memory allocated directly by the user in its code;

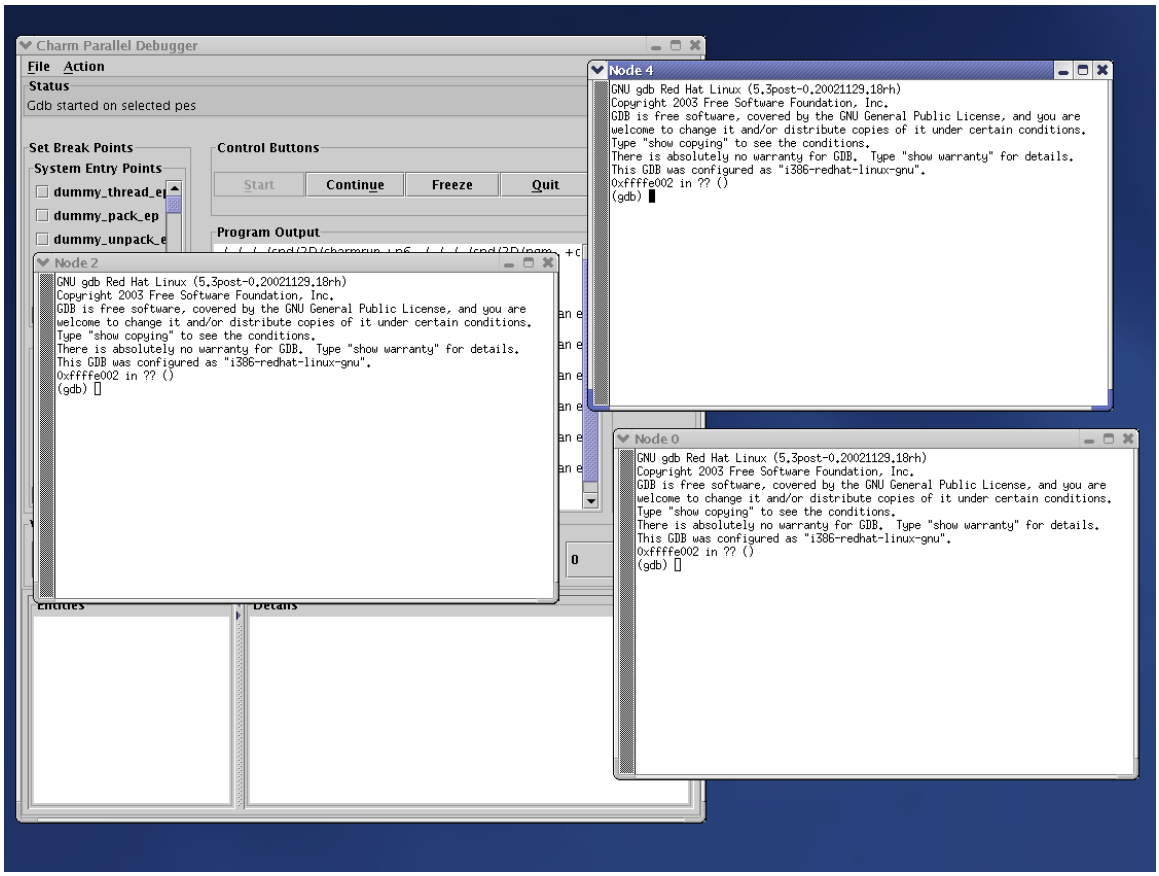

Figure 4: Parallel debugger showing instances of *gdb* open for the selected processor elements

pink memory used by messages;

orange memory allocated to a chare element;

black memory not allocated;

gray a big jump in memory addresses due to the memory pooling system, it represent a large portion of virtual space not used between two different zones of used virtual space address;

yellow the currently selected memory slot;

Currently it is not possible to change this color association. The bottom part of the view shows the stack trace at the moment when the highlighted (yellow) memory slot was allocated. By left clicking on a particular slot, this slot is fixed in highlight mode. This allows a more accurate inspection of its stack trace when this is large and does not fit the window.

Info →Show Statistics will display a small information box like the one in Figure 6.

A useful tool of this view is the memory leak search. This is located in the menu Action → Search Leaks. The processor under inspection runs a reachability test on every memory slot allocated to find if there is a pointer to it. If there is none, the slot is partially colored in green, to indicate its status of leak. The user can the inspect further these slots. Figure 7 shows some leaks being detected.

If the memory window is kept open while the application is unfrozen and makes progress, the loaded image will become obsolete. To cope with this, the "Update" button will refresh the view to the current allocation status. All the leaks that had been already found as such, will still be partially colored in green, while the newly allocated slots will not, even if leaking. To update the leak status, re-run the Search Leaks tool.

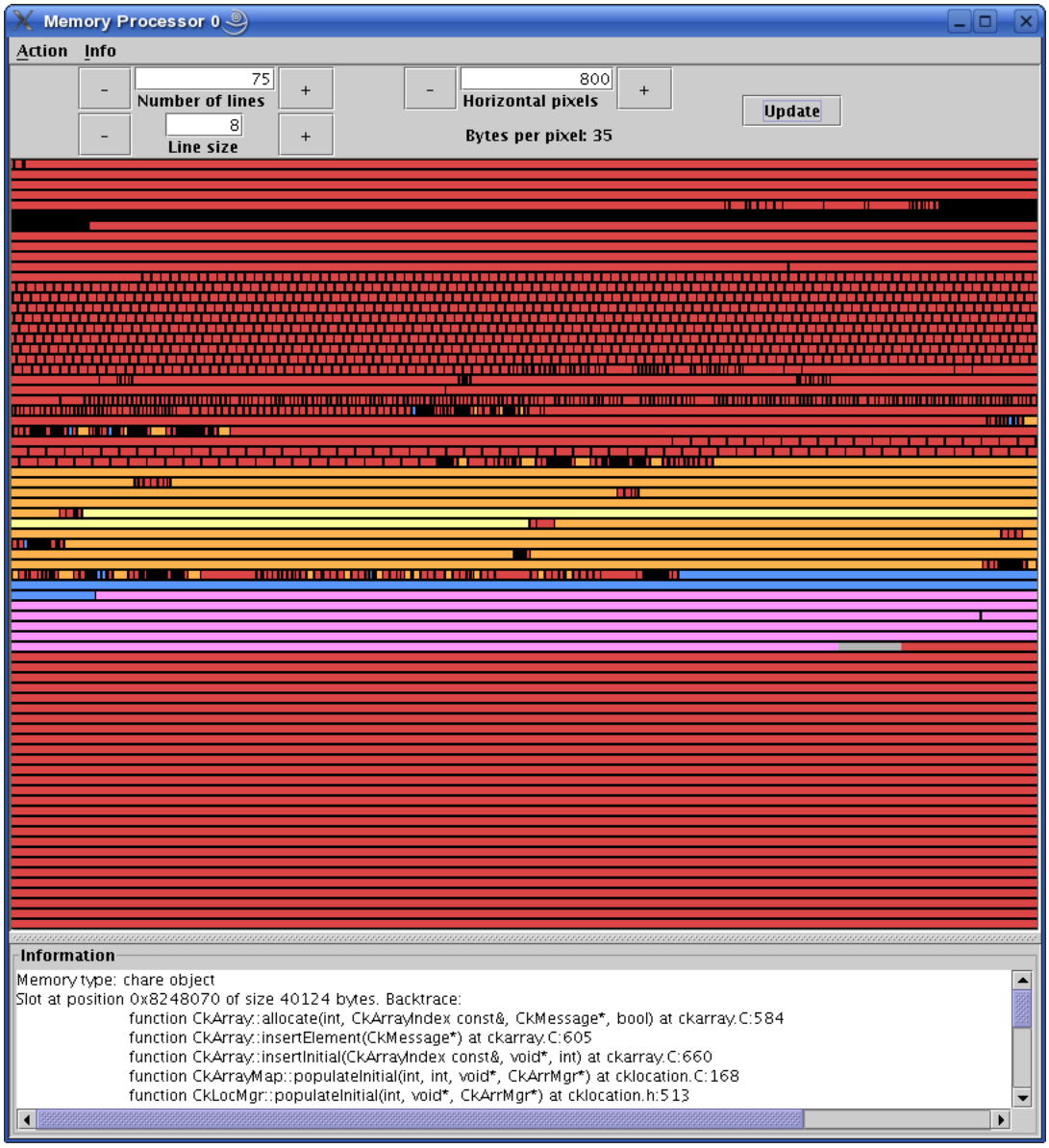

Figure 5: Main memory view

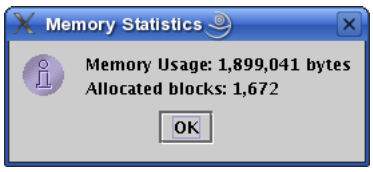

Figure 6: Information box display memory statistics

Finally, when a specific slot is highlighted, the menu Action → Inspect opens a new window displaying the content of the memory in that slot, as interpreted by the debugger (see next subsection for more details on this).

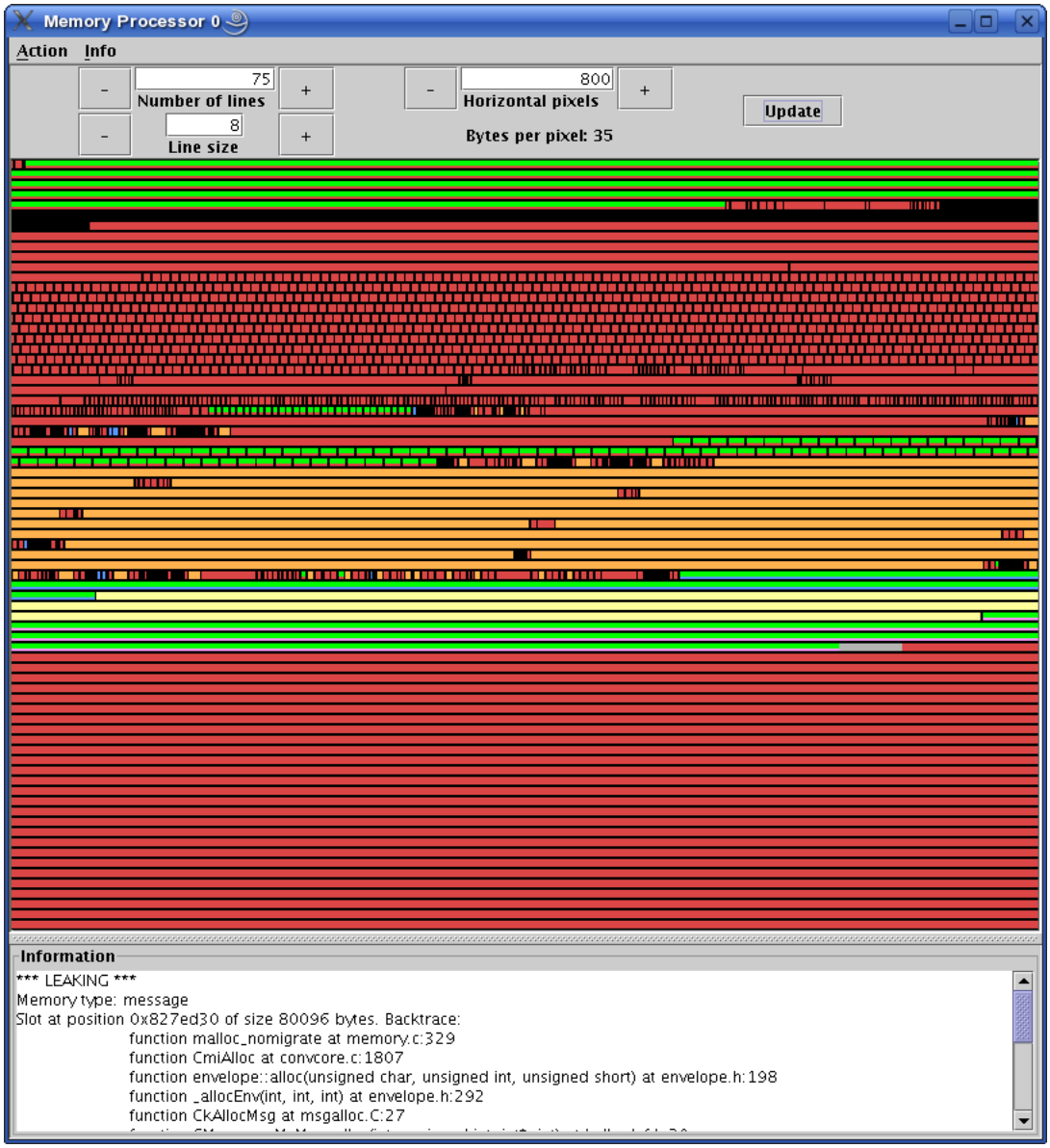

Figure 7: Memory view after running the Search Leaks tool

4.5.2 Inspector framework

Without any code rewriting of the application, CharmDebug is capable of loading a raw area of memory and parse it with a given type name. The result (as shown in Fig. 8), is a browseable tree. The initial type of a memory area is given by its virtual table pointer (Charm++ objects are virtual and therefore loadbable). In the case of memory slots not containing classes with virtual methods, no display will be possible.

When the view is open and is displaying a type, by right clicking on a leaf containing a pointer to another memory location, a popup menu will allow the user to ask for its dereference (shown in Fig. 8). In this case, CharmDebug will load this raw data as well and parse it with the given type name of the pointer. This dereference will be inlined and the leaf will become an internal node of the browse tree.

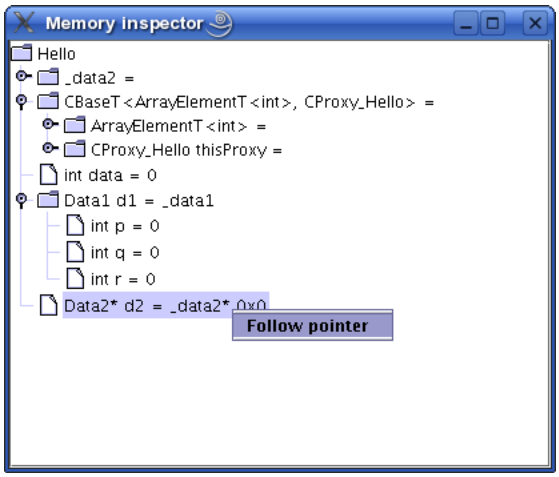

Figure 8: Raw memory parsed and displayed as a tree

5 Debugger Implementation Details

The following classes in the PUP framework were used in implementing debugging support in charm.

- `class PUP::er` - This class is the abstract superclass of all the other classes in the framework. The `pup` method of a particular class takes a reference to a `PUP::er` as parameter. This class has methods for dealing with all the basic C++ data types. All these methods are expressed in terms of a generic pure virtual method. Subclasses only need to provide the generic method.
- `class PUP::toText` - This is a subclass of the `PUP::toTextUtil` class which is a subclass of the `PUP::er` class. It copies the data of an object to a C string, including the terminating NULL.
- `class PUP::sizerText` - This is a subclass of the `PUP::toTextUtil` class which is a subclass of the `PUP::er` class. It returns the number of characters including the terminating NULL and is used by the `PUP::toText` object to allocate space for building the C string.

The code below shows a simple class declaration that includes a `pup` method.

```
class foo {
 private:
  bool isBar;
  int x;
  char y;
  unsigned long z;
  float q[3];
 public:
  void pup(PUP::er &p) {
    p(isBar);
    p(x);p(y);p(z);
    p(q,3);
  }
};
```

5.1 Converse Client-Server Interface

The Converse Client-Server (CCS) module enables Converse [**?**] programs to act as parallel servers, responding to requests from non-Converse programs. The CCS module is split into two parts - client and server. The

server side is used by a Converse program while the client side is used by arbitrary non-Converse programs. A CCS client accesses a running Converse program by talking to a `server-host` which receives the CCS requests and relays them to the appropriate processor. The `server-host` is `charmrun` [**?**] for netlrts- versions and is the first processor for all other versions.

In the case of the netlrts- version of Charm++, a Converse program is started as a server by running the Charm++ program using the additional runtime option "*++server*". This opens the CCS server on any TCP port number. The TCP port number can be specified using the command-line option "*server-port*". A CCS client connects to a CCS server, asks a server PE to execute a pre-registered handler and receives the response data. The function `CcsConnect` takes a pointer to a `CcsServer` as an argument and connects to the given CCS server. The functions `CcsNumNodes`, `CcsNumPes`, `CcsNodeSize` implemented as part of the client interface in Charm++ returns information about the parallel machine. The function `CcsSendRequest` takes a handler ID and the destination processor number as arguments and asks the server to execute the particular handler on the specified processor. `CcsRecvResponse` receives a response to the previous request in-place. A timeout is also specified which gives the number of seconds to wait till the function returns a 0, otherwise the number of bytes received is returned.

Once a request arrives on a CCS server socket, the CCS server runtime looks up the appropriate registered handler and calls it. If no handler is found the runtime prints a diagnostic and ignores the message. If the CCS module is disabled in the core, all CCS routines become macros returning 0. The function `CcsRegisterHandler` is used to register handlers in the CCS server. A handler ID string and a function pointer are passed as parameters. A table of strings corresponding to appropriate function pointers is created. Various built-in functions are provided which can be called from within a CCS handler. The debugger behaves as a CCS client invoking appropriate handlers which makes use of some of these functions. Some of the built-in functions are as follows.

- `CcsSendReply` - This function sends the data provided as an argument back to the client as a reply. This function can only be called from a CCS handler invoked remotely.
- `CcsDelayReply` - This call is made to allow a CCS reply to be delayed until after the handler has completed.

The CCS runtime system provides several built-in CCS handlers, which are available to any Converse program. All Charm++ programs are essentially Converse programs. `ccs_getinfo` takes an empty message and responds with information about the parallel job. Similarly the handler `ccs_killport` allows a client to be notified when a parallel run exits.