Mentat 2.5 User's Manual

The Mentat Research Group

Technical Report No. CS-94-06
February 17, 1994

Mentat 2.5 User's Manual

The Mentat Research Group

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
mentat@Virginia.edu

This work is funded in part by NSF grants ASC-9201822 and CDA-8922545-01, and NASA grant NGT-50970.

The following people have contributed to the Mentat project: Andrew Grimshaw, Ed Loyot, Jon Weissman, Padmini Narayan, Emily West, John Karpovich, Laurie MacCallum, Tim Strayer, Brian Paine, David Mack, Virginio Vivas, and Gorell Cheek.

Table of Contents

February 17, 1994

Mentat 2.5
User's Manual

A user's guide to the installation, utilities. and library of the Mentat object-oriented parallel processing environment.

1.0 Introduction

Mentat is an object-oriented parallel processing system designed to directly address the difficulty of developing architecture-independent parallel programs. The fundamental objectives of Mentat are to

- provide easy-to-use parallelism,
- achieve high performance via parallel execution, and
- facilitate the portability of applications across a wide range of platforms.

The Mentat approach exploits the object-oriented paradigm to provide high-level abstractions that mask the complex aspects of parallel programming, including communication, synchronization, and scheduling, from the programmer. Instead of managing these details, the programmer concentrates on the application. The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution.

The Mentat system is made up of two major components: the *Mentat Run-Time System* (RTS) and the *Mentat Programming Language* (MPL). Programs written in MPL interact with the RTS via compiler generated library calls. The RTS provides a set of services needed by application programs, including instantiation and scheduling of Mentat objects, and program graph construction and management.

MPL is an object-oriented, parallel programming language designed to support high degrees of parallelism yet be easy to use. The MPL syntax is a superset of C++, the object-oriented extension of C. Users decompose their programs into objects and "tag" some of these objects as candidates for parallel execution. This programming paradigm

not only supports the decomposition of the program into logically cohesive parts, it also facilitates software engineering principles such as simplification of large programs and code reuse.

Mentat's underlying model of computation is the Macro Data-Flow model. Data-flow analysis, performed by the MPL compiler **mplc**, supports parallel execution by detecting data dependencies and blocks only when results from parallel computations are required. The MPL compiler **mplc** translates MPL code into C++ with embedded run-time system calls. These calls allow the RTS to automatically manage all communication and synchronization.

A *Mentat network* is a collection of hosts which are available to work on an application. The network is specified by listing the host's names — along with other host-specific information — in a start-up configuration file called *config.db*. Once a Mentat network is started, users can run their Mentat applications. Using the various utilities provided, a user can also see the way the application is distributed to the participating hosts, and even monitor the progress of the application.

An active Mentat network has an Instantiation Manager (**im**) daemon and a Token Manager Unit (**tmu**) daemon on each host. The **im** handles requests to instantiate (that is, create) Mentat objects and determines on which host to place them, and performs other services involving the management of Mentat objects The **tmu** provides the token matching services required by regular (stateless) Mentat objects. When the **tmu** has collected and matched all of the tokens needed for a regular Mentat object invocation, it asks the **im** to initiate an instance of the object to service the request. It then forwards the tokens to that new instance.

This document explains in detail how to install and use the Mentat system. The installed Mentat system is superficially just another user account with directories and files. Inside the Mentat account are all of the utility and system binaries, the Mentat language libraries, man pages, and documentation. However, any user, given the proper permissions, can start and use the Mentat system; the Mentat account — although a valid user — is actually a convenient repositoriy for the Mentat system files.

Section 2 steps the reader through unpacking Mentat and verifying that the system has been properly installed. Section 3 is a tour of the Mentat directories. Section 4 overviews the Mentat user commands. Section 5 discusses how to configure a Mentat network. Section 6 describes how to use MentatView, the real-time graphical network monitor. Section 7 discusses **mplc**, the compiler for the Mentat Programming Language. Finally, Section 8 discusses the Mentat library *libmentat*.

1.1 A Mentat Bibliography

Along with the overviews given by the several Mentat manuals, detailed descriptions of the various aspects of the Mentat object-oriented parallel processing system are given in published papers and technical reports. Below is a bibliography of these papers; please refer to these for more information.

Details on the Mentat Programming Language are given in *Mentat Programming Language Reference Manual*, which is available with the standard Mentat distribution. Refer to this manual for writing Mentat applications.

The Mentat Approach to Parallel Programming

A.S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, pp. 39-51, May 1993.

The Mentat Computation Model

A.S. Grimshaw, W.T. Strayer, and P. Narayan, "Dynamic Object-Oriented Parallel Processing," *IEEE Parallel & Distributed Technology: Systems & Applications*, pp. 33-47, May 1993.

A.S. Grimshaw, "The Mentat Computation Model - Data-Driven Support for Dynamic Object-Oriented Parallel Processing," Department of Computer Science Technical Report CS-93-30, University of Virginia, May 1993.

The Mentat Run Time System

A.S. Grimshaw, J. Weissman, W.T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing," Department of Computer Science Technical Report CS-93-40, University of Virginia, July 1993.

A.S. Grimshaw, "The Mentat Run-Time System: Support for Medium Grain Parallel Computation," *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 1064-1073, Charleston, SC., April 9-12, 1990.

The Mentat Scheduler

A.S. Grimshaw and V. E. Vivas, "FALCON: A Distributed Scheduler for MIMD Architectures", *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 149-163, Atlanta, GA, March 1991.

The Mentat Communication System

A.S. Grimshaw, D. Mack, and W.T. Strayer, "MMPS: Portable Message Passing Support for Parallel Computing," *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 784-789, Charleston, SC., April 9-12, 1990.

Experiences using Mentat

A.S. Grimshaw, E.A. West, and W.R. Pearson, "No Pain and Gain! - Experiences with Mentat on Biological Application," *Concurrency: Practice & Experience*, Vol. 5, No. 4, pp. 309-328, June 1993.

P. Narayan, *et. al.*, "Portability and Performance: Mentat Applications on Diverse Architectures," Department of Computer Science Technical Report TR-92-22, University of Virginia, July 1992.

J.B. Weissman, A.S. Grimshaw, and R. Ferraro, "Parallel Object-Oriented Computation Applied to a Finite Element Problem," to appear in *Scientific Computing*, 1993.

J.F. Karpovich, M. Judd, W.T. Strayer, and A.S. Grimshaw, “A Parallel Object-Oriented Framework for Stencil Algorithms,” *Proceedings of the Second Symposium on High-Performance Distributed Computing*, Spokane, WA, July 21-23, 1993.

Mentat Approach to Heterogeneous Processing

A.S. Grimshaw, J.B.Weissman, E.A. West, and E. Loyot, “Meta Systems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems,” TR-92-43, Department of Computer Science, University of Virginia, December, 1992.

General

B. Stroustrup, “What is Object-Oriented Programming?” *IEEE Software*, pp. 10-20, May, 1988.

Margaret A. Ellis and Bjarne Stroustrup, “The Annotated C++ Reference Manual”, Addison-Wesley Publishing Company, 1990.

1.2 Mentat Distribution

There are two flavors of the Mentat system: a version for distributed workstation environment, and a mulitprocessor version for a tightly-coupled MIMD machine. The workstation architectures supported by the workstation-based version are:

- Sun 3 workstations
- Sun 4 SparcStations
- Silicon Graphic Iris workstations
- IBM RS/6000 workstation

A single Mentat network may contain hosts of any or all of these architecture types. In this way a MIMD “platform” can be created using a collection of otherwise autonomous hosts.

The multicomputer verison does not need to create a MIMD platform—it uses the one that exists within the multicomputer. This version is not currently compatible with the workstation version. The multicomputer architecture supported by Mentat is:

- Intel iPSC/860 Hypercube (Gamma)

This Mentat distribution is public and free of charge. However, since we wish to keep track of the community of Mentat users, please refer interested parties to us rather than passing along the distribution file.

Mentat is a research vehicle and as such is constantly in evolution. There are likely to be unwanted features and deficiencies within this release. If you find any mistakes or have any comments, suggestions, or questions, please do not hesitate to call or e-mail to mentat@Virginia.edu.

1.3 Changes in this Upgrade

Mentat 2.5 improves on earlier versions in several ways:

- *New directory structure*

  The directory structure of Mentat has been updated and streamlined for ease of use. A full explanation of the directories can be found in Section 3 of this document.

- *Configuration enhancements*

  The configuration of Mentat hosts has been greatly expanded. Hosts can be grouped together by processor attributes, network proximity, or other attributes for more efficient scheduling. In addition, multiple Mentat configurations can utilize the same host. A full explanation of configuration management can be found in Section 5 of this document.

- *Enhanced library*

  The Mentat library has been updated to include the classes *mstream*, *oolib*, and *DD_array* (formerly known as *app_misc*). Use of these classes requires only that the proper header files be included. A full explanation of the use of these classes can be found in Section 9 of this document.

- *Sequential persistent objects*

  A new type of Mentat object has been added--sequential persistent objects. This allows the user control over the order of execution of methods on a persistent Mentat object.

- *Create/Destroy overloading*

  The create and destroy methods for persistent Mentat objects have been enhanced to allow overloading. Users may exert more control over the placement and topology of persistent Mentat objects.

- *Inheritence*

  The Mentat Programming Language has been enhanced to allow for Mentat class hierarchies. A full explanation of Mentat class inheritance can be found in the Mentat Programming Language Reference Manual.

- *Heterogeneity*

  Mentat can now be configured to run on networks of hetergeneous processors. A full explanation of how to configure a heterogeneous network can be found in Section 5 of this document.

- *Regular object multiplexing*

  Previous releases limited the number of active regular objects to a few per processor. This release permits up to one thousand per processor.

- *Enhanced compiler*

  The MPL compiler **mplc** has been enhanced for easier use. Also, many bugs have been fixed, resulting in a far more robust compiler.

- *Bug fixes*

  We continue the on-going effort to discover and erradicate annoying “features.”

- *Additional Platform Support*

  We have added the IBM RS/6000 workstation to the list of supported platforms. We plan to add the DEC Alpha and the HP workstations soon.

1.4 Help

There are several help services available.

- *Reference manuals*

  This manual and the *Mentat Programming Language Reference Manual* are designed to provide all of the support necessary to use the Mentat system.

- *On-line manual pages*

  The Mentat system commands are in manual section 1. The Mentat language features are in manual section 3 and 3z; to find a manual entry in section 3z you must specify the section number: man 3z intro. File types are in manual section 5.

- *A Tutorial*

  The Mentat Tutorial provides a step-by-step course in starting a Mentat network and using some example applications.

- *Electronic mail to mentat@Virginia.edu*

  We invite you to ask questions and make comments about Mentat through the electronic mail service. The address mentat@Virginia.edu will put you directly in touch with the Mentat Research Group at the University of Virginia.

- *Electronic mail relay service*

  For general questions, and questions for which answers may help other users, the mentat relay is available. These questions and comments are distributed to everyone registered on the relay. To register for the relay send mail to mentat@Virginia.edu.

2.0 Installation

This section provides the step-by-step process of installing Mentat. As mentioned earlier, there are two versions of Mentat: a workstation-based version and a version designed for multicomputers. Each version is packaged in a distribution file that must be downloaded from the University of Virginia. The steps for unpacking the distribution files, once gotten, are outlined in the following two sections.

2.1 Workstation-Based Mentat Installation

The workstation-based Mentat system distribution file is available via anonymous ftp from uvacs.cs.virginia.edu in the pub/mentat directory. The file is archived and compressed. Once downloaded and unpacked, the file unfolds into the directories and subdirectories containing the various files need to run the Mentat system. The following is a list of the procedures you must follow to correctly install Mentat. You should be familiar with UNIX and UNIX directory structures. Some of the installation requires system administrator assistance.

Setting-Up the Mentat Account

All of the Mentat system support files and binaries, such as the MPL compiler and the run-time system, should be maintained in a user account called “mentat.” Have your system administrator create a user “mentat.” The instantiation managers in the Mentat RTS can use system loading information for scheduling if the Mentat account is a member of the *kmem* group. Some platforms (such as the Sun workstations) allow this; others platforms, or system administrators, may not. The penalty is that the RTS will make some naive scheduling decisions.

The Mentat account should be active for each machine on which you wish to run Mentat. It is important, however, that all the machines are part of a shared file system, and that the files in the Mentat account are visiable to each machine. (Note that it is all right to have some machines be in the *kmem* group while others are not.)

We recommend that you also create a group called “mentat” to which all users of Mentat, including the Mentat account, belong. This simplifies protection.

Step 1 Set the protections for the Mentat account first by issuing the command:

```
$ umask 002
```

and second by placing the line “umask 002” in the *.profile* or *.login* file for the Mentat account.

Step 2 Move the Mentat distribution file *mentat.tar.Z* to the home directory of the Mentat account. (We will refer to this home directory as “/users/mentat”; you may, however, supply the full path name or use “/home/mentat.”)

Step 3 To reconstitute the distribution file, issue the following command from the home directory of the Mentat account:

```
$ cat mentat.tar.Z | uncompress | tar xpBf -
```

When this is finished check the directory structure — it should look like that given in Section 3.0.

Step 4 Type the following lines at the UNIX prompt from the home directory if Mentat’s account is using *sh* or a derivative (*ksh, zsh, bash*):

```
$ MENTAT=/users/mentat
$ . $MENTAT/etc/env_sh
```

If the Mentat account’s shell is the C shell or a derivative (*csh* or *tcsh*), issue the following:

```
$ setenv MENTAT /users/mentat
$ source $MENTAT/etc/env_csh
```

Also put these lines in the Mentat account’s *.profile* or *.login* file.
The scripts *env_sh* and *env_csh* define and export several Mentat environment variables, including $MENTAT_BIN, the bin directory for system commands and utilities, $MENTAT_ARCH, the architecture type of this machine, and $MENTAT_USR_BIN, the bin directory for user applications.

Step 5 Verifiy that $MENTAT_ARCH is set properly by issuing:

```
$ echo $MENTAT_ARCH
```

This should return the architecture type of this machine. (Valid machine types are "sun3," "sun4," "sgi," and "rs6000.") You should verify that $MENTAT_ARCH gets properly set for each architecture type in your system.

Step 6 There are some permissions that do not get set properly after the unpacking of the distribution file. To set these permissions, issue:

```
$ $MENTAT/etc/set_protection
```

This commnand will warn you that "you are not a member of group kmem" if that is the case. Run this command once, after unpacking the distribution file.

Step 7 Change directory to $MENTAT/usr/bin. Copy the file *sample.config.db* to *config.db*, and edit *config.db*. Follow the instructions in *config.db*, and refer to Section 5 for more information. The "trialrun" cluster should have only a few machines (1-3) initially to verify that the system works.

Verifying Mentat

If all went well with setting up the Mentat account and setting the privileges and environment variables, then the Mentat system is ready to run applications. Now you should be able to build and execute the example applications described at the end of this document and, after reading the *Mentat Programming Language Reference Manual*, you can build and execute your own Mentat applications. Before going that far, however, first verify that the system is properly installed by following these steps:

Step 1 Change directories to $MENTAT. Verify that the account "mentat" can run a remote shell on each of the hosts in the *config.db* file by issuing:

```
$ rsh <hostname> finger
```

where "<hostname>" is the name of a remote host. Some sites "trust" local machines. If your site does not, you will need an appropriate *.rhosts* file. See your system administrator for details.

Step 2 Verify that the *config.db* file is well-formed by issuing the command:

```
$ print_config
```

Look at the output to see if you have specified the hosts and clusters properly.

Step 3 Verify all hosts in the *config.db* file will respond by issuing the command:

```
$ check_config
```

This command will ask each host if there are any Mentat processes present; for our purposes it verifies that the *config.db* file will work with utility commands, and that each host in the *config.db* file will accept an "rsh."

Step 4 To start the Mentat network described by the list of hosts in the *config.db* file, issue the following command:

```
$ start_mentat
```

This command will invoke a remote shell on each of the hosts in the *config.db* file and will start the instantiation manager (**im**) on that host. You should see messages indicating the progression of **start_mentat**. If a Mentat network using the same port number as is specified in the *config.db* file is already started on some machine in the

config.db file, **start_mentat** will report which machine and who owns the Mentat processes.

Step 5 If **start_mentat** completes without error messages, the network is running. You may verify that the Mentat system is indeed running by issuing:

```
$ list_objects
```

This will list all Mentat objects that the system is executing; in this case all of the machines should respond (indication of a correctly running instantiation manager) but no objects will be shown. The system is running properly if all hosts respond and the command returns; if some host is not properly included in the system, the command will hang just after the errant host's name is printed. You must interrupt the command with <ctrl C>, issue the **kill_mentat** command, check that no Mentat-owned processes exist on any machine listed in the *config.db* file, and then go back to Step 2 to try again.

Step 6 Run the shell script "dotest" by issuing the following command:

```
$ dotest
```

This shell script in turn calls various benchmark programs. The results of the test should look something like:

```
gauss 100 1 elapsed msec = 706
gauss 100 1 elapsed msec = 696
gauss 100 1 elapsed msec = 834
```

(etc.)

Step 7 It is not necessary or desirable to bring down the Mentat system after each application; however, there are circumstances when you will need to bring the system down. To remove all Mentat processes on the hosts in the *config.db* file, issue:

```
$ kill_mentat
```

An indication of the Mentat system's demise will be printed to the screen, and a subsequent **check_config** command will report that no Mentat processes are present on each of the machines in the *config.db* file.

Step 8 If **kill_mentat** fails to kill all of the mentat processes, or hangs in the middle, a more radical method for clearing the Mentat network may be employed. The command

```
$ purge_mentat
```

will remove all Mentat processes on all machines in the *config.db* file, regardless of who started those processes or which port number they are using. Use this command with caution.

Individual Mentat Users

Mentat can of course be run by any user in the *mentat* group. Each user must insert several Mentat-specific lines into their *.profile* or *.login* file so that the user can use the Mentat system. As with the Mentat account, the $MENTAT environment variable must be set in the user's *.profile* file as follows:

```
MENTAT=/users/mentat
. $MENTAT/etc/env_sh
```

or the user's *.login* file:

```
setenv MENTAT /users/mentat
source $MENTAT/etc/env_csh
```

Alternatively, a user wishing to run the Mentat commands in the current login session must type in the appropriate set of lines at the command prompt.

Custom Mentat Configuration

A Mentat network can be started by creating a *config.db* file in some directory other than $MENTAT/usr/bin. The custom Mentat network must be started, used, and killed, from the same directory as the custom *config.db* file is, otherwise the Mentat utility commands will look for the *config.db* file in the default location (namely, $MENTAT/usr/bin). In this way muliple Mentat networks can be running at the same time, and as long as the ports (specified in the *config.db* file—see Section 5) are different, muliple Mentat networks can have hosts in common. Use the **check_config** command to verify a custom *config.db* file before starting a custom Mentat networkt.

2.2 The Multicomputer Version Mentat Installation

To install Mentat on the Intel Gamma or Paragon, consult the document “Mentat on the Intel Gamma and Paragon”.

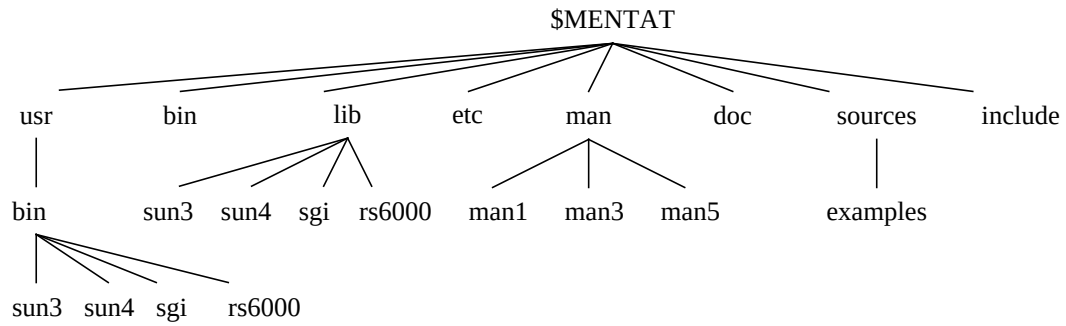
3.0 Mentat Distribution Directory Structure

The Mentat directory structure is shown in Figure 1. There are eight directories under $MENTAT, three that contain executables, one for libraries, two for help, and two with source code or include files.

$MENTAT/usr

The *usr* directory is designed to contain the user-specific files. In particular, the *bin* directory contains (1) the default *config.db* file, and (2) a subdirectory for each

Figure 1 **Mentat Directory Structure**



architecture. These architecture-specific subdirectories are the repository for application executables. If you want to run an application on a Mentat network of mixed architecture types, you must compile that application on a representative of each architecture and copy the exectuatable to $MENTAT_USR_BIN.

$MENTAT/bin

The *bin* directory (also known as $MENTAT_BIN) contains all of the Mentat utility commands.

$MENTAT/lib

The *lib* directory (also known as $MENTAT_LIB) contains the Mentat library *libmentat*. There are two directories under each of the several architecture-specific subdirectories, att and gnu. There is an appropriately constructed library libmentat for each architecture and compiler. (Actually, for some platforms we do not support AT&T C++, so there is only a GNU version of the library under that architecture's directory.)

$MENTAT/etc

The *etc* directory contains some useful Mentat scripts, such as *set_protection*, as well as the environment scripts *env_sh* and *env_csh*.

$MENTAT/man

The *man* directory contains the manual pages for on-line help with the Mentat system. There are three subdirectories. The *man1* subdirectory contains manual pages describing the various Mentat uitlity commands. The *man3* subdirectory contains manual pages describing the language and library classes. The *man5* subdirectory contains manual pages describing the various files used by the Mentat run-time system such as the configuration file.

$MENTAT/doc

The *doc* directory contains the postscript of all of the documentation incuded with the Mentat distribution, including this manual, the *Mentat Programming Language Reference Manual*, and the *Mentat Tutorial*.

$MENTAT/sources

The *sources* directory contains all of the source code included with the Mentat distribution, specifically, the examples source code in the *examples* subdirectory. The Mentat Tutuorial wil step you through some of these examples.

$MENTAT/include

The *include* directory contains the various header files that the Mentat Programming Language MPL uses.

4.0 Mentat User Commands

Any user with properly set Mentat system environment variables can issue the Mentat commands to start, stop, or monitor the Mentat system. The set of user commands are listed below, along with some explanation of the use of these commands.

Many of the command use the file *config.db* to get information about the Mentat network configuration. The commands look in the current working directory for a file named *config.db*; failing to find one there, they look in the directory $MENTAT/usr/bin.

Some user commands do not make sense in a multiprocessing environment; these commands are noted.

NOTE: These commands use the contents of the file *config.db*; if any command is run in a directory with a *config.db* file other than the one used to start the Mentat network, unpredictable results occur.

Check_Config

***Synopsis*: `check_config`**

The **check_config** command checks each host in the *config.db* file to determine if Mentat processes (the **im** or **tmu**) are present and are using the same port as specified in the *config.db* file. If such an **im** process is present on some host, **check_config** reports who started Mentat on that host. If Mentat is unstable on some host, **check_config** asks if it should clean up any dead processes on that host. This command is useful for confirming that the hosts in a *config.db* file do not conflict with a Mentat network already in place.

Check_Host

***Synopsis*: `check_host [host_list]`**

The **check_host** command will check each host in the argument list for the presence of *any* Mentat processes (the **im** or **tmu**), and report if they are present. This command is useful for quickly checking the status of arbitrary hosts without first creating a *config.db* file.

Not available for multicomputer version.

Kill_Mentat

***Synopsis*: `kill_mentat`**

The **kill_mentat** command removes all Mentat processes — that is, the instantiation manager **im** and all of its child processes — from the Mentat network described by the *config.db* file.

Kill_Mstream

***Synopsis*: `kill_mstream`**

The **kill_mstream** command is used to remove processes created by the use of mstreams (named mfileobj and mfilesys) in the event that the user program fails and the mstream processes do not get destroyed. A kill_objects will NOT remove these processes since they are owned by the user and not mentat. This command should be run on the machine from which the main program was run.

Not available for multicomputer version and for use with g++.

Kill_Objects

***Synopsis*: `kill_objects -a`**
`kill_objects -c` *`class_name`*
`kill_objects -h` *`host_name`* `[-a | -p` *`object_pid`*`]`
`kill_objects -n` *`host_num`* `[-a | -p` *`object_pid`*`]`

The **kill_objects** command kills the specified Mentat objects while Mentat is running. When a Mentat application is aborted, some Mentat objects may be left running. By issuing the **kill_objects** command, these objects can be removed without disturbing the whole Mentat system.

Issuing the **kill_objects** command with the **-a** flag alone causes all Mentat objects on all hosts in the *config.db* file to be killed, including all objects that exist due to someone else's application. *Use this with caution; you may inadvertently kill other user's objects.*

The **-c** *class_name* flag kills all objects with the specified class name.

The **-h** *host_name* and **-n** *host_num* flags indicate a specific host, either by string or by number (the number is the order in which the host appears in the *config.db* file). Once a host is specified additional flags indicate which objects to kill: the **-a** flag kills all Mentat objects on that host, or the **-p** *object_pid* flag kills only the specified object.

Once the **kill_objects** command has been issued, peer objects attempting to communicate with killed objects will hang.

Mentat must be running when this command is issued; the command will hang if an instantiation manager (**im**) is not present on a host for which objects are being listed.

List_Objects

***Synopsis*: `list_objects [-a | -h` *`host_name`* `| -n` *`host_num`* `]`**

The **list_objects** command lists the active Mentat objects for the Mentat network as defined by the *config.db* file. Mentat objects are created by a running Mentat application. If no arguments are specified, **list_objects** will list all active Mentat objects, including the objects of other users of this Mentat system. When the **list_objects** command is issued with the **-h** *host_name* option, only objects of that specific host are listed. The Mentat system can also refer to hosts by number; when issued with the **-n** *host_num* option, only objects of that specific host number are listed.

Semantically, regular Mentat objects (see the *Mentat Programming Language Reference Manual*) are instantiated only for the purpose of executing a called member function. When the member function has completed, the process executing the regular Mentat object is killed. This is inefficient for multiple calls on member functions of a regular Mentat object. Consequently, the Mentat RTS keeps bookkeeping information about the object in place, even after the member function call completes. To see all Mentat objects, including these idle objects, issue the **list_objects** command with the option **-a**.

Mentat must be running when this command is issued; the command will hang if an instantiation manager (**im**) is not present on a host for which objects are being listed.

Print_Config

***Synopsis*: `print_config`**

The **print_config** command parses the *config.db* file and prints out the configuration specified. This command is a good way to verify that the *config.db* file is well-formed, and that the hosts have the attributes intended. The **print_config** command searches for the *config.db* file first in the current working directory, then in the $MENTAT/usr/bin directory.

Purge_Mentat

***Synopsis*: `purge_mentat`**

The **purge_mentat** command removes all Mentat processes from all machines in the *config.db* file. The **purge_mentat** command differs from the **kill_mentat** command in that it removes all instantiation managers (**im**) and their children, without regard to which port the Mentat network described by the *config.db* file was using.

Resume_Mentat

***Synopsis*: `resume_mentat -a`**
`resume_mentat -h` *`host_name`*
`resume_mentat -n` *`host_num`*

The **resume_mentat** command reenables the scheduler to place Mentat objects on the named host. The **list_objects** command will show the status of each host; **resume_mentat** changes the status of a host from SUSPENDED to RUNNING. The **-a** option resumes placement of Mentat objects on all hosts. The **-h** *host_name* option resumes placement of Mentat objects on host *host_name*. The **-n** *host_num* option resumes placement of Mentat objects on host number *host_num*.

Mentat must be running when this command is issued; the command will hang if an instantiation manager (**im**) is not present on a host for which objects are being listed.

Not available for the multicomputer version.

Start_Mentat

***Synopsis*: `start_mentat [-n` *`nice_value`*`]`**

The **start_mentat** command creates and initializes a Mentat system. The **start_mentat** command uses the *config.db* file (either in the current working directory or, failing to find one here, in the $MENTAT/usr/bin directory) to determine the number and names of the hosts to be included in the Mentat network. The command then checks the suitability of each of the hosts listed. If Mentat is already running on a particular host using that port number, or if some other process is using the port number Mentat uses, a message will indicate which host is suspect and, if possible, who started Mentat on that host. If all hosts are suitable for starting a Mentat network, **start_mentat** invokes a remote shell on each of the hosts listed. Each remote shell issues commands to create an instantiation manager (**im**) process on that host.

If **start_mentat** fails for some other reason, you must first remove all Mentat processes from the machines where start_mentat was successful.

The **-n** *nice_value* option provides the mechanism for starting Mentat processes with a particular nice value.

Standard I/O is not well defined in a distributed system. For Mentat, the files *stderr* and *stdout* of all remote processes are bound to the window in which **start_mentat** is executed. Because all Mentat objects are children of one of the remote shells, they all share the same *stdout*. Separate input/output streams can be realized using the *mstream* facility (see Section 8.0).

Suspend_Mentat

***Synopsis*: `suspend_mentat -a`**
`suspend_mentat -h host_name`
`suspend_mentat -n host_num`

The **suspend_mentat** command will disallow the scheduler from placing Mentat objects on the named host. Issuing the **list_objects** command will show the status of each host; **suspend_mentat** changes the status of a host from RUNNING to SUSPENDED. The **-a** option suspends placement of Mentat objects on all hosts. The **-h** *host_name* options suspends placement of Mentat objects on host *host_name*. The **-n** *host_num* option suspends placement of Mentat objects on host number *host_num*.

Mentat must be running when this command is issued; the command will hang if an instantiation manager (**im**) is not present on a host for which objects are being listed.

Not available for multicomputer version.

5.0 Configuring Mentat

When a Mentat network is started there are several pieces of information necessary to ensure that all hosts in the system have the proper run-time system processes instantiated, and that these processes have values for their various configuration parameters. This section describes the Mentat configuration process, and how it uses either default or user-specified configuration values.

Workstation-Based Configuration

In the workstation-based Mentat, hosts may be partitioned into *families* and, orthogonally, into *clusters* (Figure 2). A family of hosts shares the same attributes such as processor type and MIPS rating. A complete list is given below.

While families represent processor attributes, clusters indicate locality information that is used in the location policy during object scheduling. Members of a cluster are presumed by the **im**'s (instantiation managers) to be closer together, e.g., they may reside on the same network segment. The **im**'s exploit this information and try to schedule new objects on the same cluster as the creator (unless the creator has specified via a location hint that the object is to be placed far away). If an idle node cannot be found in the local cluster, the object is transferred to another cluster.

Currently, the capabilities of different host types are not exploited by the scheduler. Instead, the hosts are treated uniformly for scheduling purposes. With respect to the types of hosts that can comprise a configuration, we are currently limited to either homogeneous sets of workstations, or to workstations that have the same data

representation and alignment characteristics, e.g., Sun 4's, IBM RS 6000's, and Silicon Graphics.

The Configuration Database

The configuration database *config.db* contains all of the information needed to configure Mentat at run-time. It contains the names of all of the hosts, family information, cluster information, scheduling policy specification and parameters, and other pertinent data. The configuration database is a text file that may be edited as the configuration changes. Database information is specified using a simple configuration language. The database is read and parsed at start-up by utilities such as **start_mentat**, by application main programs, and by the instantiation manager (**im**) and token matching unit (**tmu**). Because there may be multiple configuration databases, Mentat first looks in the current working directory and, if a *config.db* is not found there, it looks in the directory $MENTAT/usr/bin.

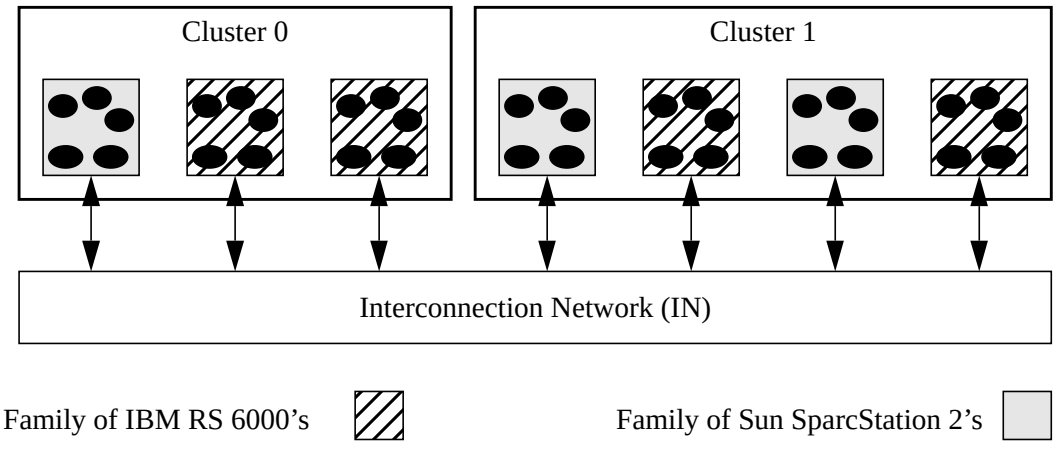
Parts of a Configuration File

There are four major parts to a configuration file, host specification, family specification, cluster specification, and global modifiers. The first three parts must appear in the file in this order; the global modifiers may appear anywhere in the file. In the first part each host is specified and described using this syntax:

HOST <*hostname*> { *modifier_list* }

Every host in the Mentat network must be described in this way. The value of <host-name> should be the name of the host as returned by issuing the "hostname" command

Figure 2 **Families and Clusters of Hosts**



on that host; that is, it may be necessary to have a fully-qualified host name exactly as "hostname" returns, so the message passing system can find the proper host addresses.

Once all of the hosts are properly described, the hosts can be grouped into families, using the following syntax:

FAMILY <*family_name*> { *host_name_list modifier_list* }

The use of a family specification to further describe the attributes of the hosts is optional. Further, it is not necessary to put each host into a family. However, any given host may belong to only one family at a time. The family names are arbitrary.

A useful family to create is the "inactive" family. By having as an attribute ACTIVE NO, no host in the family is used in the Mentat network. Such a family is useful for rapidly adding and deleting hosts from a Mentat network while testing.

The last part of the configuration file is the cluster specification, which requires the following syntax:

CLUSTER <*cluster_name*> { *host_name_list modifier_list* }

Every host specified with a HOST keyword must be included in one and only one cluster. There may be as many clusters as is necessary to partition the hosts, up to one cluster per host. Usually, clusters describe nearness attributes, such as having a common subnetwork. The cluster names are arbitrary.

There are four global modifiers that apply to the entire configuration:

IM_PORT *port_number*
TRANSFER_LIMIT *limit*
LOCATION_POLICY *policy*
TRANSFER_POLICY *specifier*

The modifier IM_PORT specifies a port number to be used by the **im**. The default port for the **im** is 1686. By specifying a unique port, multiple Mentat networks may be running on any given host, each using a different configuration file. The value *port_number* must be an integer greater than 1500.

The modifier TRANSFER_LIMIT specifies the maximum number of hosts/clusters that will be examined when scheduling. The value *limit* is an integer.

The modifier LOCATION_POLICY is used to specify the how the scheduler decides where to place objects. Valid *policy* values are:

ROUND_ROBIN
BEST_MOST_RECENTLY
RANDOM, or
PROCESSOR_POWER

ROUND_ROBIN indicates that the next host to try when scheduling is selected using a round-robin policy. BEST_MOST_RECENTLY is a random location policy where the last hop is used to transfer the task to the best (in terms of a threshold value) host seen so far. RANDOM places objects randomly in the Mentat network. PROCESSOR_POWER indicates that objects are placed on processors in order of decreasing MIPS rating.

The modifier TRANSFER_POLICY specifies whether to transfer the task to a different host or execute it locally. The transfer policy must be:

LOAD *threshold*, or
RUN_QUEUE *threshold*.

The LOAD transfer policy is based on the CPU load of the host. If the CPU load is greater than the specified threshold, the task will be transferred. The value *threshold* is an integer indicating the greatest tolerable load. The RUN_QUEUE transfer policy is based on the run queue length of the host. If the queue length is greater than the specified *threshold*, the task will be transfered.

Modifiers

The modifier list that can be included in the host, family, and cluster specifications is built from the following modifiers:

- ACTIVE *yes_or_no*

  Default is YES. If NO, Mentat will not use the host or hosts specified by the HOST, FAMILY, or CLUSTER.

- MENTAT_OBJECTS *threshold*

  The *threshold* value of Mentat objects allowed on the host or hosts specified. If more than *threshold* objects are on a host, then the transfer policy (default, or specified by TRANSFER_POLICY) will determine whether to transfer the tasks.

- USE_KERNEL_MEMORY *yes_or_no*

  This modifier indicates whether kernel memory should be used when making transfer policy decisions on the host or hosts specified. The default is NO. YES is supported only for Sun 3's and Sun 4's. In order to use kernel memory, Mentat must be in the group *kmem* (see your system administrator).

- TMU_MEMORY *size*

  The value *size* indicates maximum amount of memory that the **tmu** may use to store tokens, in 1024 byte units, for the host or hosts specified. Once this limit is reached, the **tmu** caches tokens in the scratch directory on disk. The default is 2048 (2 megabytes).

- MAX_IDLE_REGULAR *number*

  This modifier indicates the maximum number of idle regular objects on the host or hosts specified. Higher values of *number* improve the performance of codes that use many different regular object classes, but consume more virtual memory. The default is 2.

- TYPE *host_type*

  This modifier indicates the processor type of the host or hosts specified. The value of *host_type* must be one of SUN3, SUN4, RS6000, or SGI.

- FLOPS *number*

  This modifier indicates the number of millions of floating point instructions per second attainable for the host or hosts specified.

- MIPS *number*

  This modifier indicates the number of millions of instructions per second attainable for the host or hosts specified.

- NUM_PROCS *number*

  This modifier specifies the number of processors on the host or hosts specified in the case of multiprocessors such as the Sun Sparc 10, or multicomputers such as the Intel iPSC/860 Gamma.

- SYSTEM_BIN path_name

  This indicates the full path name where Mentat system binaries are located. The value *path_name* is a literal string. The default is $MENTAT_BIN.

- USR_BIN *path_name*

  This indicates the full path where user executable object binaries are located. The value *path_name* is a literal string. The default is $MENTAT_USR_BIN.

- SCRATCH_PATH *path_name*

  The directory where system scratch files should be located. The value *path_name* is specified as a literal string. The default is /tmp.

- VIRTUAL_NODE_SIZE *number*

  On multicomputers (e.g., the iPSC/860 Gamma) this modifier indicates the number of processors per virtual node. The minimum is 4. Larger numbers make more efficient use of processor resources, but at the risk of the IM or TMU becoming a bottleneck. If the application does not dynamically create objects very often, then large numbers (32, 64, 128) are good choices. The *number* must be a power of 2.

A sample configuration file is given in Figure 3. Note that comments can be placed anywhere; any text following "//" is treated as a comment. Seven hosts are specified here. There are three machine types, two of which are described in families, the other is described in the appropriate HOST specifier. By coincidence, the hosts of the family IPC happen to be close to each other; they are grouped as a cluster. The same is true for the acc_RS6000 hosts, and the hosts of type SGI. Global modifiers specify the location policy, the transfer policy, and the transfer limit. Since no IM_PORT modifier is present, the default of 1686 is used.

Multicomputer Version Configuration

Since the Intel Gamma NX allows only one process per processor, a different process mapping paradigm is adopted. If each Gamma node ran a copy of the instantiation manager, then no user processes could be scheduled on the Gamma. Instead, the allocated Gamma sub-cube is partitioned into sets of virtual nodes comprised of a number of processors. Each virtual node has a single instantiation manager responsible for scheduling within the virtual node, and a token matching unit. All virtual nodes contain the same number of processors and the user may select the number of processors at configuration time. The first two processors of each virtual node are reserved for the instantiation manager and token matching unit. Therefore, the minimum virtual node size is four. Typically there are at least sixteen processors in a virtual node. When an object instantiation request is accepted within a virtual node, the instantiation manager executes a "load" onto one of the virtual node processors. If the

Figure 3 **A Sample *config.db* File**

```
// Comments may appear anywhere
HOST elf { }
HOST acacia { }
HOST palm {
   SCRATCH_PATH “/bigtmp”
   // host has a large attached disk.
}
HOST disney {TYPE SGI}
HOST sutherland {TYPE SGI }
HOST holmes { }
HOST watt { }
// Generally all host entries should appear
// before all cluster and family entries.

FAMILY acc_RS6000 {
   holmes watt
   TYPE RS6000
   USR_BIN “/home/mentat/usr/bin/rs6000”
   SYSTEM_BIN “/home/mentat/bin/.rs6000”
}

FAMILY IPC {
   elf
   palm
   acacia
   USE_KERNEL_MEMORY YES
}

CLUSTER one {
   elf palm acacia
TRANSFER_LIMIT 2
}

CLUSTER two { holmes watt }
CLUSTER three { disney sutherland }

LOCATION_POLICY RANDOM
TRANSFER_POLICY LOAD 30 // If the load is over 30%,
                        // transfer the task
TRANSFER_LIMIT 1 // Sets the inter-cluster
                 // transfer limit to 1,
                 // rather than the default of 2
```

request is not accepted within the virtual node, the request is passed on to the instantiation manager of another virtual node.

6.0 MentatView

It is often difficult to observe the execution of a parallel or distributed system because there is often no global knowledge maintained about the state of each node. Even if global knowledge is available, there is still the problem of how to display this information in a way that is useful and understandable. MentatView is a monitor for the Mentat run-time system that addresses this problem by allowing Mentat users to display various system performance parameters at run-time. The data are displayed as simple bar graphs that are constantly updated as the system executes. This information can be used by the user for debugging or by the system manager to see if the Mentat RTS is making good scheduling decisions, and to see the impact of the Mentat processes on the nodes in the system.

MentatView shows the user which nodes are in the current Mentat configuration, how many Mentat objects are on a given node, and what the load and queue sizes are for a given node. A sample MentatView session is shown in Figure 4. In general, MentatView provides a way to monitor various system performance parameters while the system is running. Keep in mind that running MentatView can seriously impact system performance; it generates quite a bit of message traffic and uses all of the CPU cycles available on the host on which it is running.

MentatView runs using the Mentat runtime system on a network of Sun workstations running UNIX and X Windows. It is not available for the mulitcomputer version of Mentat.

To run MentatView, perform the following steps:

Step 1 Have X Windows running and ensure that the Mentat run-time system is running.

Step 2 Enter the command:

```
$ mentatview -display hostname:0.0
```

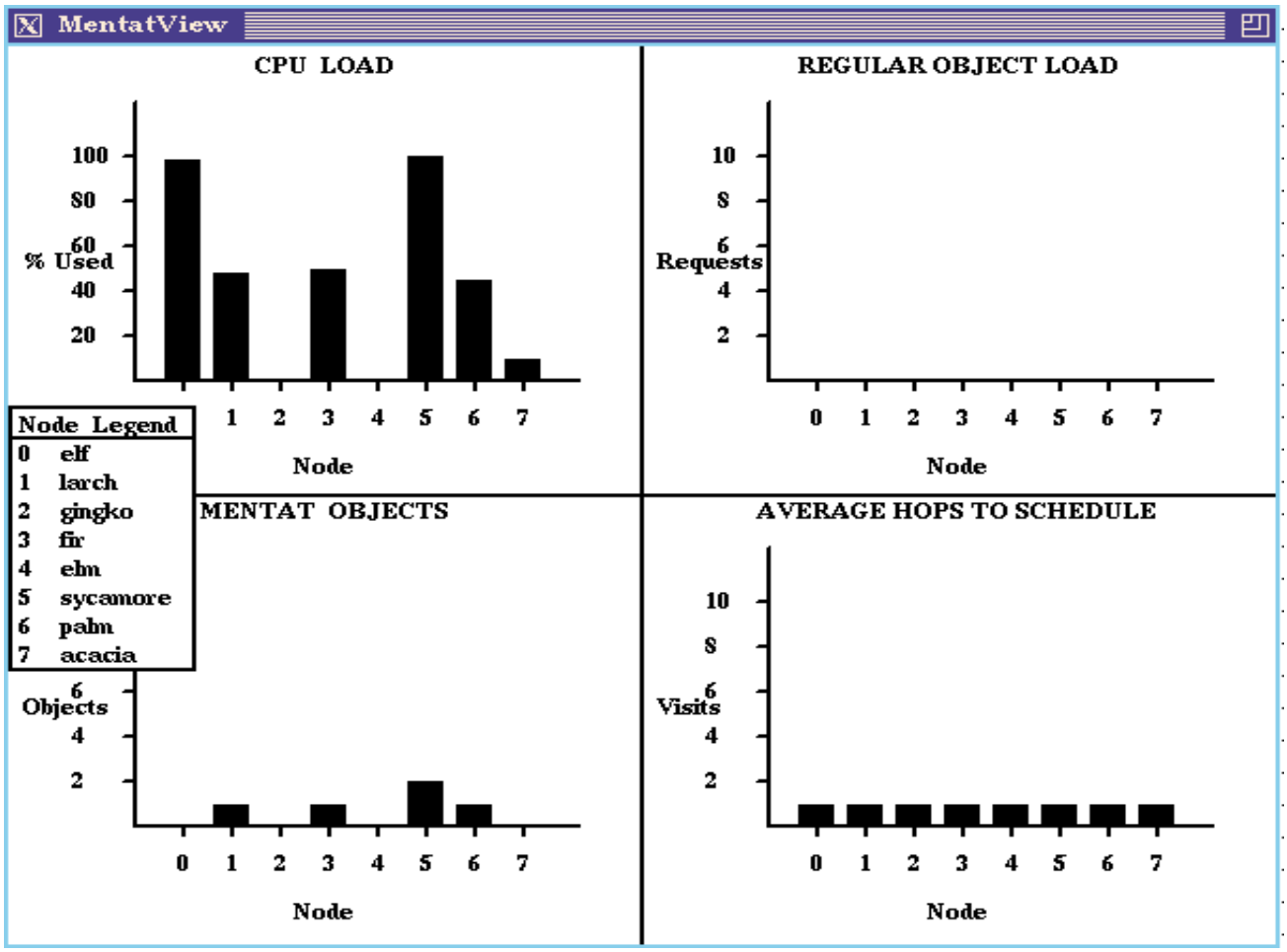
where the *hostname* is the name of the machine on which MentatView's window is to appear. The display flag is optional if the $DISPLAY variable is defined; MentatView's window will appear on the machine on which it is being executed in this case.

Step 3 Once the MentatView window appears on the screen, a pop-up menu will appear when any mouse button is pressed with the pointer in the MentatView window. To select a menu command, position the pointer on the command and press any mouse button. The commands and their meanings are listed below:

Objects

not yet implemented.

Figure 4 **Sample MentatView Display**



(Un) Freeze

This command will stop the updating of graphs in the MentatView window, allowing the user to freeze an instant of time. This is a toggle command. Selecting it again unfreezes the MentatView window. This keeps MentatView from generating network traffic that can perturb performance.

Node Legend

This command pops up a window that matches the node numbers used in the MentatView graphs with a hostname. To close the window, place the pointer in the window and press any mouse button.

Quit

This option exits MentatView. This is the only safe way to exit MentatView. Using <ctrl-C> or the X window kill command will cause the Mentat RTS to hang.

Step 4 When done using MentatView, click on the **Quit** option in the MentatView menu.

7.0 Mentat Programming Language Compiler

Programs written in MPL are compiled by the Mentat Programming Language compiler **mplc**. The compiler **mplc** is actually a UNIX shell script that parses the command line to determine the MPL program filename and any command line switches. The MPL program filename must have a ".c" suffix. **Each Mentat class definition must be contained in a separate file, and each Mentat class must be compiled separately.** The mplc script then manages a series of parsing and code generation passes. First, the MPL code is passed through the C preprocessor, removing all includes and defines. The preprocessed code is then piped into the MPL front-end, called *mplfront*. The mplfront parses the MPL code, removing the Mentat keywords and producing C++ code with the appropriate run-time system calls. The result is a translation file whose name has a ".trans.c" suffix. For example, an MPL program in a file called "test_matrix.c" is compiled into a C++ program placed in a file called "test_matrix.trans.c." Compilation is then completed when **mplc** invokes a C++ compiler on the translation file. By default, the translation file is removed at the end of compilation.

NOTE: mplc supports several C++ compilers, including GNU's g++; due to problems with GNU, higher versions than g++ 2.2.2 will not compile.)

The mplc Command

The options to **mplc** are a superset of the C++ compiler options. All C++ compiler-specific options are accepted by **mplc** and passed on to the appropriate phase.

The syntax for the **mplc** command is:

mplc

***Synopsis*: `mplc` [-A*arch*] [-C*compiler*] [-C++] [-c] [-D*name*[=*def*]]**
[-E] [-I*pathname*] [-inherit] [-L*directory*]
[-nocompile] [-n] [-Q] [-trans] [-v] [*cpp options*]
[*C++ compiler options*] [-o *target*] *sourcefile*
[*objectfiles*] [-l*library*]

where:

-A*arch*

Specifies architecture type (the default architecture type is gotten from $MENTAT_ARCH; this option overrides that value for cross compiling).

-C*compiler*

mplc supports three C++ compilers: g++, CC, and CC_saber; this flag specifies which one to use. The default compiler used is CC_saber. Unsupported native C++ compilers can also be used, but some defaults may not be set properly. NOTE: **mplc supports g++ version 2.2.2 only; due to problems with GNU higher versions of g++ will not compile.**

-C++

Run only the C++ compiler leaving intact all of the **mplc** defaults. This is useful for compiling serial versions of Mentat applications.

-c

Suppress linking and produce only a *file***.o** file.

-D*name*[=*def*]

Define a symbol name. This is equivalent to a #define directive in the source. If no *def* is given, name is defined as '1'.

-E

Stop at the C++ preprocessing stage. The output of **mplc** will be preprocessed C code only (directed to *stdout* unless a **-o** *target* is specified).

-inherit

Suppress linking and produce a *file***.inherit** file; this file is then linked into another Mentat class binary file to facilitate inheritance. The class in *file***.inherit** is the base class.

-I*pathname*

Add pathname to the list of directories in which to search for #include files with relative filenames (not beginning with slash '/'). The preprocessor searches for #include files in the directories named with **-I** options (if any) before searching in the standard include directories.

-l*library*

Link with object library *library* (for **ld**(1)).

-L*directory*

Add directory to the list of directories containing object-library routines (for linking using **ld**(1)).

-n

No-execute mode: shows what **mplc** would do but does not execute.

-nocompile

Causes mplc to stop processing after the MPL front-end stage. If the **-trans** option is not specified, the intermediate file will be deleted and **mplc** will not produce output. The default is to complete compilation.

-o *target*

Name the result of the compilation target.

-Q

Override default libraries, include files, and bindings used by mplc upon the invocation of the C++ compiler. The user must supply all include path, library path, and definition binding information, including: the path to the Mentat header files ($MENTAT/include), the path to local versions, if any, of standard header files, the path to standard header files, the path to the Mentat libraries ($MENTAT_LIB/<compile type>, where compiler type is “att” or “gnu”), and the link option for the Mentat library (*libmentat.a*).

-trans

Keep the intermediate file *file***.trans.c**, which is the output of the MPL front-end processing. This is file consists of C++ code. The default is to remove it after compilation.

-v

Verbose mode: show explicitly what **mplc** is doing.

Makefiles

Often a *makefile* is used to manage this compilation process. An example *makefile* is given in Figure 5. The `MFLAGS`, `CFLAGS`, and `LFLAGS` variables indicate **mplc** flags, C flags (such as defines and include file path names), and link flags (such as link file path names). Note that the result of the compilation is copied to the $MENTAT_USR_BIN directory—this is where the Mentat run-time system looks for the executable of a class when a member function is invoked.

The compiler **mplc** includes some defaults for libraries, include files, and binding when invoking the C++ compiler. These can be overridden with the -Q option. The include file path includes the standard compiler header file directory as well as the Mentat header file directory $MENTAT/include. Some standard include files have, strictly speaking, illegal C++ syntax; where modifications were necessary, the changed files were placed in the $MENTAT/include/local directory structure, and modifications were documented in README files.

Errors Generated by mplfront

The script **mplc** manages the various passes required to compile MPL code. The first pass, *mplfront*, translates MPL code into valid C++ code. To do this, mplfront must completely parse the MPL code; the mplfront parser is more conservative than various C++ parsers and as a consequence, valid MPL code may generate error messages. These error messages can be disconcerting; the messages that indicate real errors usually include the words “parse error,” and the “make” will fail.

Figure 5 **A Typical Makefile for a Mentat Application**

```
MPLC = mplc
MFLAGS = -Cg++
CFLAGS = -O
LFLAGS = -static -lm
HEADERS =
OBJECTS =
TARGET_DIR = $(MENTAT_USR_BIN)

############################################################

all: gauss sblock

clean:
   rm -f gauss sblock *.trans.*

gauss:   $(HEADERS) gauss.c sblock.h $(OBJECTS)
         $(MPLC) $(MFLAGS) $(CFLAGS) gauss.c \
            -o gauss $(OBJECTS) $(LFLAGS)
         strip gauss
         cp gauss $(TARGET_DIR)

sblock:  $(HEADERS) sblock.c sblock.h $(OBJECTS)
         $(MPLC) $(MFLAGS) $(CFLAGS) sblock.c \
            -o sblock $(OBJECTS) $(LFLAGS)
         strip sblock
         cp sblock $(TARGET_DIR)
```

Known Bugs

The following are the known bugs in the current version of mplfront. Some of them are very rare and have not been seen for a long time; nonetheless we mention them in the event that you encounter something similar.

- The *Annotated C++ Reference Manual*[1] states that inline member functions declared inside of a class are not type checked until the complete class declaration has been seen (p. 171). However, mplfront currently does type checking as the inline member function is parsed. This often causes spurious errors, particularly with some of the standard header files, like “stream.h.” To correct this problem, mplfront must be rewritten to do type checking as a separate pass, after the initial parsing

1. Margaret A. Ellis and Bjarne Stroustrup, “The Annotated C++ Reference Manual”, Addison-Wesley Publishing Company, 1990.

phase. To work around the problem, write the member function outside of the class using the inline keyword.

- Conversion functions (p. 272) cause a syntax error. The mplfront is built around a public domain C++ grammar.[2] There is an error in the current version of this grammar that makes it unable to parse legal conversion functions. A new version of the grammar has been developed, but the new version has not been incorporated into this version of mplfront. There is currently no way to work around this problem. Note the local versions of some header files (like *stream.h*) may have to be made with the conversion functions commented out or surrounded by an `#ifdef`.

Unsupported C++ Features

The following features of C++ are not supported.

- As per the *Annotated C++ Reference Manual*, p.405, the keyword **overload** is no longer supported.
- Extra semicolons outside of the function scope will cause parse errors. These extra semicolons constitute empty declarations and are not supported as in the *Annotated C++ Reference Manual*.
- Although MPL allows for inlined functions, the compiler does not perform an inline expansion. A normal function call is made.
- Member variables of a Mentat class may not be public, as per the MPL language specification (see the *Mentat Programming Language Reference Manual*).
- Operators may not be overloaded in Mentat classes.
- Note that some standard header files still use the notion that classes, structs, unions and enums defined inside another class, struct or union, have global scope. This is not the case as described in the *Annotated C++ Reference Manual*. This may cause parse errors when compiling the standard header files. The offending code must be modified or commented out.

MPL Language Features As Yet Unsupported

The following features of MPL have not yet been implemented:

- Location hints. Although these are not currently implemented, the effect of the location hint *co-locate* can be achieved as follows. There are actually two create() functions defined. The first is create() (with no arguments) which works as described in the language manual. The second, create(&mentat_object) will create a new Mentat object on the same node as the Mentat object passed as an argument. This achieves the same effect as using co-locate.

2. The public domain C++ grammar is copyright 1989, 1990 by James A. Roskind (jar@-florida.HQ.Ileaf.COM).

Bug Fixes Pending

- Constructors/Destructors for Mentat classes will not work—use `void initialize()`, `void cleanup()` member functions to get the desired effect.
- Passing object references (i.e. `&obj`) will pass a fixed-sized argument, even if tahe class has a `size_of()` member defined.
- Assigning Mentat class variables to Mentat objects returned from a Mentat class member function (by pointer) will not resolve properly, and the assignment will not be performed. To work around this, the Mentat object must first be assigned to a temporary variable. The temporary variable should then be assigned to the Mentat variable.
- Default arguments to Mentat member functions are disallowed.
- The **mplc** compiler is being integrated with newer versions of GNU's g++.

How To Get Help

If you are having difficulty using the compiler or if you have found a bug contact mentat@virginia.edu. Please include a detailed description of the problem you are having, and include a copy of the offending code and the makefile you are using. A daytime phone number would also be helpful. Any comments or suggestions would be appreciated.

8.0 Mentat Library Classes

In addition to the run-time system's functionality, the Mentat library *libmentat* contains classes that are useful in applications development. The public interface to these classes is given in several header files found in $MENTAT/include. The classes include:

- The Object-Oriented Library (*oolib.h*)
- The Mentat Stream Facility (*mstream.h*)
- The Array and Vector Classes (*DD_array.h*)
- The Sparse Vector Class (*sparse_vector.h*), and
- The File Interface for Array and Vector Classes (*DD_array_file.h*)

8.1 The Object-Oriented Library

The Object-Oriented Library provides those classes that have almost universal applicability but are not part of C++ or MPL. The class definitions reside in *oolib.h* in the $MENTAT/include directory. In particular, classes include:

- `mentat_timer`: simple CPU stopwatch timer, useful for program timings
- `list_element`: ordered generic list class
- `cell_collection`: ordered collection of tagged list_elements
- `hash_cell_collection`: hash table class with unbounded bucket size
- `queue`: FIFO queue class

- `string`: variable-size strings
- `mem_handle`: general memory allocator
- `transportable_list`: dynamic variable-size memory-contiguous list

Mentat Timer

`int overflow()`

The `mentat_timer` class implements a simple stopwatch. The member function

```
void start()
```

begins the stopwatch. When the member function

```
void stop()
```

is called, the stopwatch stops. The elapsed time is retrieved via the

```
unsigned long msec()
```

(milliseconds) and the

```
unsigned usec()
```

(microseconds) member function calls. If the `usec()` call would overflow, the member function

```
overflow()
```

will return 1; otherwise 0. The member function

```
unsigned long get_time()
```

returns the current time in microseconds.

List Element

The `list_element` class implements an ordered collection of cells. The member function

```
void append(list_element *entry)
```

appends a new entry to the back of the list. The member function

```
void clear()
```

clears the list of all elements. The member functions

```
list_element *get_next()
list_element *get_prev()
```

return the next or previous entry in the list. The member function

```
void insert(list_element *entry)
```

puts a new entry at the front of the list. The member function

```
void remove()
```

deletes the element from the list.

Cell Collection

The `cell_collection` class implements an ordered collection of cells. The constructor

```
cell_collection(int size = DEFAULT_TCELLS)
```

takes an optional argument for the maximum number of cells in the collection. The default is the defined constant `DEFAULT_TCELLS`. The member function

```
cell* insert(cell *tokp)
```

puts a new cell `tokp` into the collection. The member function

```
cell* find(cell *ctag)
```

locates the cell `ctag` by tag. The member function

```
cell* get_next()
```

returns the next cell in the collection. The member function

```
cell* get_first()
```

returns the first cell in the collection. The member function

```
void remove(cell *ctag)
```

deletes the cell `ctag` from the collection. The defined constant `EMPTY_CELL` designates a nonexistent cell pointer.

Hash Cell Collection

The `hash_cell_collection` class implements a hash table with unbounded bucket size. The constructor

```
hash_cell_collection(int size = MaxEntries)
```

takes a optional argument for the maximum number of cells in the collection. The default is the defined constant `MaxEntries`. The member function

```
cell* insert(cell *tokp)
```

puts a new cell `tokp` into the hash table. The member function

```
cell* find(cell *ctag)
```

locates the cell `ctag` by tag. The member function

```
cell* get_next()
```

returns the next cell in the hash table. The member function

```
cell* get_first()
```

returns the first cell in the hash table. The member function

```
void remove(cell *ctag)
```

deletes the cell `ctag` from the hash table. The member function

```
virtual int hash()
```

returns the hash value of the cell `item`. The defined constant `EMPTY_CELL` designates a nonexistent cell pointer. The defined constant `EMPTY_BUCKET` designates a nonexistent `cell_bucket` pointer. The member function

```
int full()
```

returns the 1 if the table is full, 0 otherwise.

Queue

The `queue` class implements a simple first in, first out queue of pointers. The constructor

```
queue(int size = P_Q_SIZE)
```

takes an integer argument size indicating the largest the queue will ever get. The default is the defined constant `P_Q_SIZE`. The member function

```
void enq(void* item)
```

enqueues an item. The member function

```
void* deq()
```

returns the item from the head of the queue. The member function

```
int count()
```

returns the number of elements in the queue.

String

The `string` class implements a variable-size string object. The overloaded constructors

```
string(int size)
string(char* src)
```

takes either an integer size or a null-terminated character array `src` as its argument. The member function

```
unsigned size_of()
```

`r`eturns the size of the `string` object.

Memory Handler

The `mem_handle` class implements a general memory allocation class for doing one's own memory management. The constructor

```
mem_handle(int size_of_objects)
```

takes an integer `size_of_object` as an indication of how much memory to acquire. The destructor

```
~cell_collection()
```

It frees all acquired memory when the object leaves scope. The member function

```
void *get_one()
```

procures a block of memory of the size specified in the constructor. The member function

```
void return_one(void *object)
```

frees the memory previously held by the structure object.

Transportable List

The `transportable_list` class implements a dynamic variable-size class that is useful for building flattened structures such as lists. An advantage to using the `transportable_list` class is, as its name implies, that the data structures can be package in such a way as to aid parameter passing between member functions of Mentat objects. The constructor

```
transportable_list(unsigned max_count,
                   unsigned object_size)
```

takes two arguments: the maximum number of items in the list, and the size of each item. The member function

```
char* append(char* mo)
```

appends the character pointer mo to the current list. The member functions

```
unsigned get_count()
unsigned get_max_count()
```

return the number of elements in the list and the maximum number of elements allowed, respectively. The member function

```
unsigned size_of()
```

returns the size of the `transportable_list` object. The member functions

```
unsigned get_flags()
void set_flags(unsigned val)
```

get and set flags for the list.

There are two operators

```
transportable_list& operator=(transportable_list& nl)
char& operator[](unsigned i)
```

for assignment and indexing.

8.2 The Mentat Stream Facility

The Mentat I/O system, `mstream`, is based on C++ streams. Default I/O is not provided by the Mentat environment. Mentat classes that require I/O have to explicitly open and manipulate I/O streams. The class definition is given in *mstream.h*.

The Mentat stream facility ensures that I/O operations performed by class instances executing on remote nodes will be performed on the application host via the local file system (i.e. in the "point-of-launch" environment), and the I/O system guarantees that file access is performed using the access privileges of the user. For example, tty I/O will be performed on the host console (as expected) instead of on a remote console.

This facility is not available for the multicomputer version of Mentat and is not available for use with g++.

```
void MENTAT_OPEN_FS(mfilesys mfs);
void MENTAT_CLOSE_FS(mfilesys mfs);
void MENTAT_ENABLE_IO(mfilesys mfs);
void MENTAT_DISABLE_IO(mfilesys mfs);

mcout <<
mcin >>
mcerr >>

class mfilebuf : public streambuf {
public:
   mfilebuf(mfilesys fs);
   mfilebuf(int nfd);
   mfilebuf(mfilesys fs, int nfd);
   ~mfilebuf();
   int close();
   mfilebuf* open(char *name, int om);
   int overflow(int c = EOF);
   int underflow();
   void setup(mfilesys fs);
};
```

Enabling I/O

If a Mentat class requires I/O, the header file *mstream.h* must be included in the source file associated with that Mentat class. There are four library routines that provide the basic stream I/O capabilities:

MENTAT_OPEN_FS
MENTAT_CLOSE_FS
MENTAT_ENABLE_IO, and
MENTAT_DISABLE_IO

For an application to correctly use the Mentat I/O system, several steps must be followed. First, the main program is responsible for instantiating the Mentat file system via a call to MENTAT_OPEN_FS(mfs), and for shutting down the file system via MENTAT_CLOSE_FS(mfs), where mfs is an unbound instance of Mentat class mfilesys. During the open call, mfs is bound to the file system object and becomes the application handle for the file system. This is only done once. Second, every application class that performs I/O must be explicitly passed mfs via some access member function. And finally, once the file system is instantiated, every class that wishes to perform I/O (including the main thread) must call MENTAT_ENABLE_IO(mfs) first. When I/O operations are finished, call MENTAT_DISABLE_IO(mfs).

I/O Interface

If the application desires to do simple tty I/O only, then the class object need only invoke MENTAT_ENABLE_IO(mfs) (and MENTAT_DISABLE_IO(mfs) at the end). After executing this routine, the standard streams mcout, mcin, and mcerr become available to the application. These are functionally identical to the C++ streams cout, cin, and cerr except that the I/O is performed in the point-of-launch environment. Once an object (or main program) enables I/O, the standard streams (mcout, mcin, and mcerr) are available to any method or subprogram within that object. The Mentat I/O system provides an interface that supports user-defined file I/O, if more general I/O is desired. In keeping with the C++ streams convention, user-defined streams must be attached explicitly to a stream buffer (C++ class streambuf). The stream buffer manages the underlying byte streams by buffering incoming and outgoing I/O. The stream buffer interacts directly with the underlying file system by performing unbuffered UNIX reads and writes on the appropriate file descriptors. The I/O operations are assumed to execute in the running environment with the privileges of the client process. To preserve point-of-launch semantics, we define a Mentat stream buffer class (mfilebuf) which is functionally identical to streambuf except that it accesses the underlying I/O system in the host environment.

The Mfilebuf Class

To create user-defined streams, the user must first define the associated stream buffer object of type mfilebuf which takes the object mfs as a constructor argument:

```
mfilebuf mybuf(mfilesys mfs)
```

This ensures that mybuf will execute its underlying I/O operations in the host environment. The stream buffer is then bound to a specific file for I/O via the open call:

```
mybuf.open(char* file, open_mode)
```

To open a file for reading use in, for writing use out, and for write-append use app. Once a stream buffer has been bound to a specific file via open (the byte source/sink is now known) I/O can be performed directly on the stream buffer object:

```
mybuf.put(char c)
mybuf.get(char c)
```

In fact, any operation on stream buffers supported by the C++ stream library is allowed.

User-defined streams are constructed by attaching the stream buffer to a stream object in the following way:

```
istream in_stream(&mybuf)
ostream out_stream(&mybuf)
```

The standard stream insertion operators can be used:

```
in_stream >> out_stream <<
```

Once the I/O operations are complete, the stream buffer needs to be closed explicitly (the user MUST remember to close or the program could have termination errors):

```
mybuf.close ()
```

One important caveat: inclusion of the standard stream header <stream.h> might cause compiler and parse errors. This is explained in the MPL Reference Manual in the Section “Unsupported C++ Features,” and a way to work around this is described.

Exceptions

Since we have defined a rather narrow interface for file I/O, the number of exceptional conditions are few. This is fortunate since there is very little error reporting should streams be misused. The most common problem is that the open on the mfilebuf fails for some reason or that a file close operation was omitted. A potential problem is that the main thread may execute `MENTAT_CLOSE_FS` while the application class objects are still running and performing I/O operations. The application must ensure that it is safe to do the `MENTAT_CLOSE_FS` by forcing the main thread to wait on application objects that are performing I/O. This can be done via strict functions and appropriately placed `rtf`’s (see MPL Reference Manual) to force a synchronization where the close follows all I/O operations. If this does not work, the user may omit the `MENTAT_CLOSE_FS` but must remember to kill the associated UNIX process (named `mfs`) when finished.

8.3 The Array and Vector Classes

The header file *DD_array.h* provides the interface for array and vector classes, and the header file *DD_array_file.h* provides the interface for many of the same classes stored in files. The interface for using sparse vectors is given in *sparse_vector.h*. The array and vector classes were developed specifically to overcome some of the problems of using matrices in a distributed memory environment; namely, the data structures are explicitly placed in contiguous memory, with member functions to aid in the decomposition and reconstruction of the structures. Three types of classes are provided:

- Two Dimensional Arrays
- Dense Vectors
- Sparse Vectors

Two Dimensional Arrays

The C++ classes `DD_floatarray`, `DD_doublearray`, `DD_intarray`, and `DD_chararray` provide an interface for two-dimensional arrays consisting of elements of type single precision floating point, double precision floating point, integer, and character, respectively. Because they are stored contiguously in memory, they are convenient to use when objects of these classes are passed as parameters to Mentat member functions.

Each array has associated with it three values: the number of rows, the number of columns, and a status code. The status code, except for a few noted instances, is for use by the programmer for setting programmer-defined error conditions. The standard linear algebra operations—addition and multiplication of two arrays, transpose, and addition, multiplication, and subtraction by a scalar value—are member functions of these classes. Further, these class provide a broad range of methods to decompose an array into several subarrays.

In the following discussion remember that the C convention numbers array indices from zero to the dimension minus one. We are only explicitly showing the methods for one of the types of DD_arrays, the `DD_floatarray`; except where noted, analogous methods exist for `DD_chararray`, `DD_doublearray`, and `DD_intarray`.

Creating a Two Dimensional Array Instance

The constructor

```
DD_floatarray(DD_floatarray_file* file_class_name)
```

creates a new array stored in the file represented by `file_class_name`., where `file_class_name` is an instance of the class `DD_floatarray_file` described below. The file represented must have previously been opened for either `READ` or `READ_AND_WRITE`. In the event that the file has not been properly opened, an array of size 0 and status -1 is created.

The following sequence of statements creates a `DD_floatarray` object:

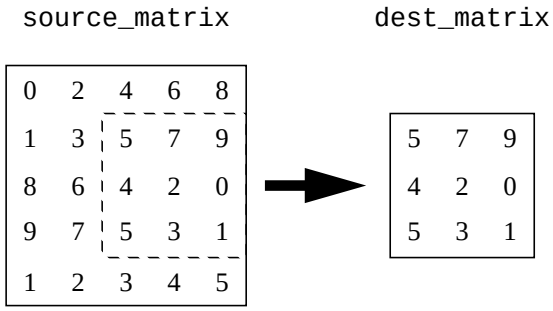
```
DD_floatarray_file* file_instance;
DD_floatarray* new_array;
file_instance =
      new DD_floatarray_file((string*)file_string, READ);
new_array = new DD_floatarray(file_instance);
```

The constructors

```
DD_floatarray(int rows, int cols)
DD_floatarray(int rows, int cols, float* ptr)
```

create an array instance having `rows` rows and `cols` columns. In the former case, the array is zero-filled. In the latter case, the array is filled with the values stored at address `ptr` in row-major order.

Figure 6 **Example of Method** `extract_region()`




```
dest_matrix = source_matrix->extract_region(1,2,3,4);
```

Creating an Instance of DD_array from an Existing Instance

The constructor

```
DD_floatarray(DD_floatarray* src)
```

creates an array instance which duplicates the array pointed to by `src`. An equivalent constructor exists for the class `DD_doublearray`. The method

```
DD_floatarray* extract_region(int ul_row, int ul_col,
                              int lr_row, int lr_col)
```

extracts a region of the array bounded by [`ul_row`, `ul_col`] on the upper left corner, and [`lr_row`, `lr_col`] on the lower right corner of the source matrix. The resulting region is returned as a `DD_floatarray`.[3] Figure 6 shows an example of such a decomposition.

In the event that the boundaries specified are out of range in the source matrix, the array is returned with zeros filled into the rows and/or columns that are out of range.

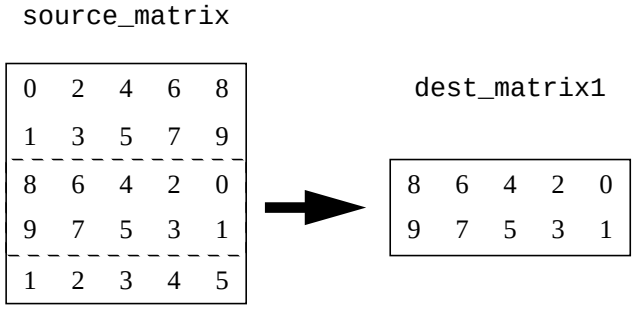
The methods

```
DD_floatarray* extract_piece_by_columns(int pieces,
                                        int the_piece)
DD_floatarray* extract_piece_by_rows(int pieces,
                                     int the_piece)
```

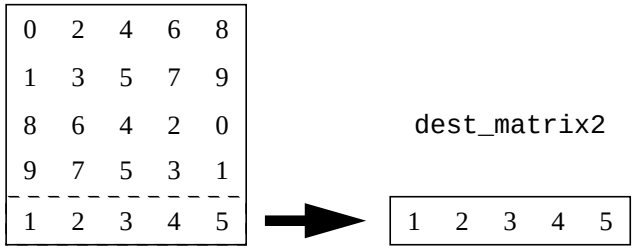
decompose the array into `pieces` different `DD_floatarray`s. The decomposition may be column-wise or row-wise. The methods return the `the_piece` piece from the

3. Note that the order of the arguments has been changed. In prior releases, the column index was specified first.

Figure 7 **Example of Method** `extract_piece_by_rows()`




```
dest_matrix1 = source_matrix->extract_piece_by_row(3,1);
```




```
dest_matrix2 = source_matrix->extract_piece_by_row(3,2);
```

decomposition. If the number of columns or rows in the array does not divide evenly, the excess columns or rows are added to each of the pieces, beginning with the first. The desired part of the decomposition is returned as a `DD_floatarray`. Figure 6 displays an example of such a decomposition.

Accessing Array Elements

The method

```
float* get_r_ptr(int row_num)
```

returns a pointer to the element in row `row_num`, column 0. The following sequence of statements demonstrates how the pointer returned can be indexed to access the elements of a particular row:

```
float element = source_matrix->get_r_ptr(1)[3];
/* If source_matrix is defined as above,
   then element = 7. */
```

The method

```
float& element(int row, int column)
```

returns a reference to element in row `row`, column `column`. The following sequence of statements demonstrates how this method can be used to access and change values in an array:

```
float value =source_matrix->element(0,4);
/* If source_matrix is defined as above,
   then value is equal to 8 */
source_matrix->element(0,4) = -4.44;
/* Now, the element at [0,4] equals -4.44 */
```

The method

```
float* operator[num_row][num_col]
```

returns a reference to the element at [`row_num`, `col_num`] in the array. This is illustrated by:

```
float value = (*source_matrix)[0,4];
/* If source_matrix is defined as above,
   then value is equal to 8 */
```

Accessing the State of an Array

The methods

```
int get_status()
int set_status(int stat)
```

manipulate the status of the array. The `get_status` call returns the status of the array. The set_status call updates the status of the array with status `stat`. The value of `stat` must be of type integer. The status is most often used as a means of returning programmer-defined error codes.

The methods

```
int num_col()
int num_row()
```

return the number of columns and rows of an array, respectively. The method

```
int same_dimensions(DD_floatarray* arg)
```

returns `TRUE` if the current instance and *arg* have the same number of rows and columns, otherwise, it returns `FALSE`. The method

```
int is_square()
```

returns `TRUE` if the array has an equal number of rows and columns, `FALSE` otherwise.

Decomposing Arrays

One of the most useful aspects of the class `DD_array` is the ability to decompose the array into several subarrays. These subarrays are also memory contiguous, and can thus serve as arguments to Mentat methods.

The methods

```
DD_floatarray** decompose_by_column(int pieces)
DD_floatarray** decompose_by_row(int pieces)
```

decompose the source array into `pieces` arrays, row-wise or column-wise, as shown in Figure 6. The methods return an array of pointers to each of the subarrays, in their respective orders within the source array. The methods

```
DD_floatarray** cyclic_decompose_by_column(int pieces)
DD_floatarray** cyclic_decompose_by_row(int pieces)
```

decompose the source array into `pieces`. Each column or row is distributed, in order, among each of the subarrays as shown in Figure 6. An array of pointers to each of the subarrays is returned.

The method

```
DD_floatarray** decompose_by_block(int row_pieces,
                                   int col_pieces)
```

decomposes the source array into `row_pieces` × `col_pieces` subarrays. The source array is decomposed first as in a row decomposition, dividing the array into row_pieces pieces. A column decomposition is then performed creating the appropriate number of pieces. An example of this decomposition is shown in Figure 6. An array of pointers to the newly created subarrays is returned. The subarrays are ordered by column decomposition order within row decomposition order.

Array Manipulation Methods

The methods

```
void overlay_region(int ul_row, int ul_col,
                            DD_floatarray* data)
void overlay_region(int ul_row, int ul_col,
                            sp_dvector* data)
```

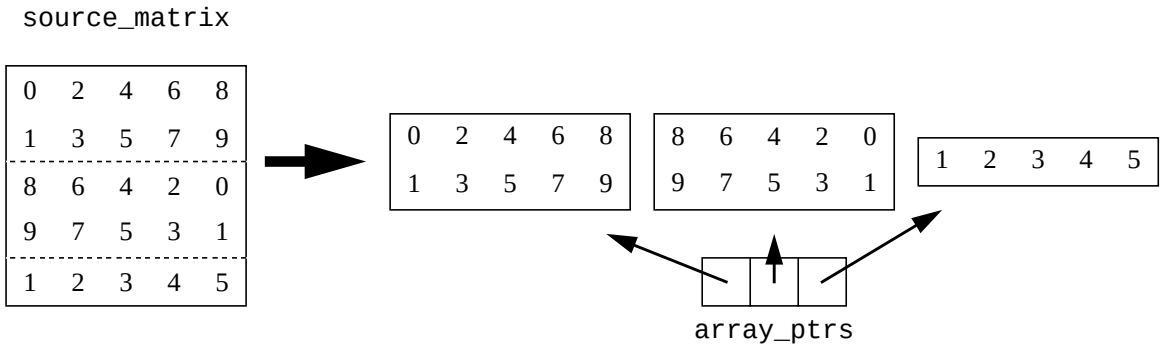
copy the array or vector `data` onto the source array starting at row `ul_row`, column `ul_col`. In the event that data is too large to be completely copied, only those regions of data are copied onto the source. This is an in place operation, therefore nothing is returned.[4] The method

```
DD_floatarray* transpose()
```

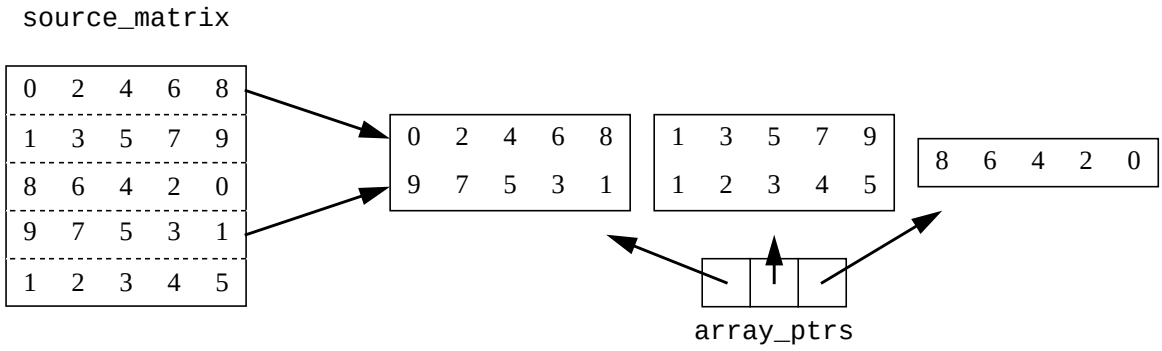
returns the transpose of the source array. The method

4. Note that the order of the arguments has been changed. In previous releases, the column index was specified first.

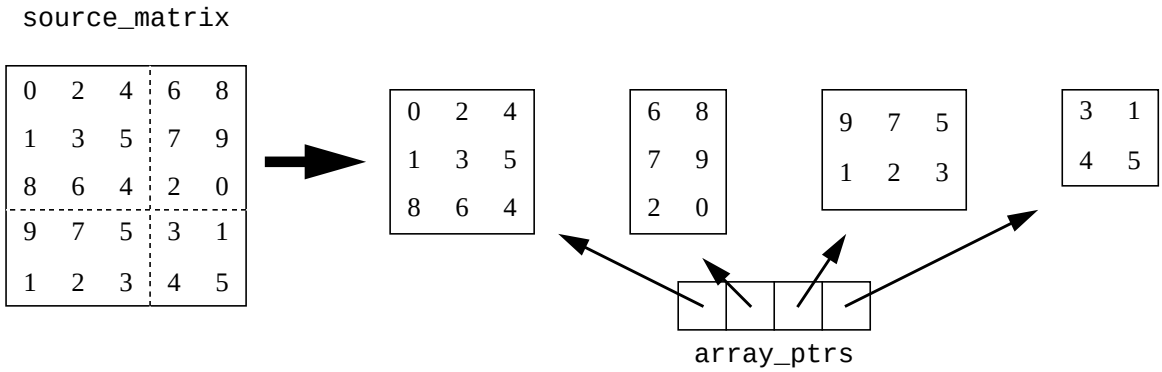
Figure 8 **Example of Methods for Array Decomposition**




```
array_ptrs = source_matrix->decompose_by_row(3);
```




```
array_ptrs = source_matrix->cyclic_decompose_by_row(3);
```




```
array_ptrs = source_matrix->cyclic_decompose_by_block(2);
```

```
void print()
```

prints the contents of the source array to standard output. For a Mentat application, this would be the window in which the Mentat system was started.

Mathematical Functions

The operators

```
void operator += (float scalar)
void operator += (DD_floatarray* data)
```

perform in-place addition of either a scalar value `scalar` or the array `data`. The following statement increments each array element by 3.4:

```
(*source_matrix)+=3.4;
/* This statement adds the value 3.4
   to all of the array elements */
```

Equivalent operators exist for the classes `DD_doublearray` and `DD_intarray` only.

The operator

```
void operator -= (DD_floatarray* data)
```

performs in-place subtraction of `data` from the source array. An equivalent operator exists for the classes `DD_doublearray`, and `DD_intarray`.

The operator

```
DD_floatarray operator *= (float scalar)
```

performs in-place multiplication of the source array with `scalar`. An equivalent method exists for the classes `DD_doublearray`, and `DD_intarray`.

The operators

```
DD_floatarray* operator * (DD_floatarray* data)
sp_dvector* operator * (sp_dvector* data)
```

perform matrix-matrix multiply and matrix-vector multiply of the source matrix and `data` and returns the result.

In the event that the arrays are not compatible (the number of columns of the source does not equal the number of columns of the destination), a 0×0 array is returned with a status of -1. Equivalent operators exist for the classes `DD_chararray`, `DD_doublearray`, and `DD_intarray` each of which return a character, double, or integer pointer, respectively.

Dense Vectors

When manipulating arrays, there is often a need to manipulate vectors as well. The classes `sp_dvector`, `dp_dvector`, and `int_vector` have been developed for this purpose. An instance of `sp_dvector` represents a single precision floating point dense vector. Likewise, an instance of `dp_dvector` represents a double precision

floating point dense vector. An instance of `int_vector` represents an integer vector. With each dense vector is associated the number of elements and a status code.

While we illustrate dense vectors using the `sp_dvector` class, for most of the methods there are equivalent constructor for the classes `dp_dvector` and `int_vector`.

Creating a Dense Vector Instance

The constructor

```
sp_dvector(sp_dvector_file *file_class_name)
```

creates an instance of sp_dvector where the elements are stored in the file represented by `file_class_name`. This method is analogous to the method

```
DD_floatarray(DD_floatarray_file *file_class_name)
```

The constructors

```
sp_dvector(int cols)
sp_dvector(int cols, float *ptr)
```

create an instance of `sp_dvector` where `cols` is the number of elements in the vector. The latter method initializes the vector with the values stored at address `ptr`. These methods are analogous to the corresponding methods for the class `DD_floatarray`.

All vectors are created as row vectors. To create a column vector, the constructor needs to be followed with a `transpose` call (described below).

Creating an Instance of Dense Vector from an Existing Instance

The method

```
sp_dvector* duplicate()
```

creates a new vector that is a duplicate of its source. The method

```
sp_dvector* extract_subvec(int first, int last)
```

extracts a subvector from the source vector starting with the element at `first` and ending with element at `last`. An equivalent method exists for the classes `dp_dvector` and `int_vector`. The method

```
sp_dvector* extract_piece(int pieces, int the_piece)
```

extracts a portion of the source vector to create a new vector. This method is analogous to the method `row_decompose()` for the class `DD_floatarray`.

Accessing Vector Elements

The method

```
float* get_vec()
```

returns a pointer to the first element in the vector. Elements of the vector can be accessed by indexing from this pointer as with the method `get_r_ptr()` in the class

DD_floatarray. An equivalent method exists for the classes dp_dvector and int_vector, each of which return a character, double, or integer pointer, respectively.

Accessing the State of a Dense Vector

The methods

```
int is_row_vec()
int is_col_vec()
```

returns TRUE if the vector is a row (or column) vector, FALSE otherwise. The method

```
int num_elements()
```

returns the number of elements contained in the vector.

Decomposing Dense Vectors

The methods

```
sp_dvector **decompose(int pieces)
sp_dvector **cyclic_decompose(int pieces)
sp_dvector *cyclic_reorder(int num_pieces)
```

decompose the vector in the same manner as the analogous methods decompose DD_arrays. There are equivalent methods for the classes dp_dvector and int_vector.

Dense Vector Manipulation Methods

The method

```
void transpose()
```

performs an in-place transpose of the vector. The method

```
void overlay_region(int offset, sp_dvector *data)
```

overlays the elements of data starting at element index offset. The source vector is modified, so there is no return value.

The method

```
void print()
```

prints the elements of the vector to standard output.

Mathematical Functions

The operators

```
void operator += (float arg)
void operator += (sp_dvector *arg)
void operator *= (float arg)
```

perform scalar addition, vector-vector addition, and scalar multiplication, respectively. The method

```
float dot_product(sp_dvector *arg)
```

performs the dot product of the source and `arg`. The result is returned.

Sparse Vectors

The classes `sp_sparsevector` and `dp_sparsevector` have been provided as a means of manipulating sparse vectors. These classes represent single and double precision floating point floating point sparse vectors. The key member variables of these classes are the dimension and the number of elements. The number of elements is the maximum number of non-zeros being represented in the vector. While we illustrate the various methods using the `sp_sparsevector` class, equivalent methods exists for the class `dp_sparsevector`.

Creating a Sparse Vector Instance

The constructor

```
sp_sparsevector(int dim, int num_elements)
```

creates an instance of `sp_sparsevector` of dimension `dim`. The integer `num_elements` denotes the maximum number of non-zero vector elements that will be associated with the vector: the vector may contain at most `num_elements` elements whose indices range in value from 0 to `dim`.

The constructor

```
sp_sparsevector(int dim, int num_elements,
        int* index_ptr, float* data_ptr, int num_vals)
```

creates an instance of `sp_sparsevector` with dimension `dim`. The vector is initialized with the elements stored at address `data_ptr` with their corresponding indices stored at `index_ptr`. The integer `num_vals` indicates the number elements that are stored at `data_ptr`. The integer `num_elements` need not equal `num_vals`.

In the event that `num_vals` does not reflect the number of elements with which the vector is to be initialized, the behavior is non-determinant.

Creating an Instance of Sparse Vector form an Existing Instance

The method

```
sp_sparsevector* extract(int first_index,
                         int last_index)
```

creates a new vector whose dimension and number of elements is that of the source. The elements of the new vector are those from the source between the indices `first_in-`

dex and last_index. In the event that the boundaries are out of range, a vector of dimension 0 and status -1 is returned.

Accessing Sparse Vector Elements

The method

```
int* get_index_ptr()
```

returns a pointer to the first index of the vector, and the method

```
float* get_vec_ptr()
```

returns a pointer to the first value within the vector. The operator

```
float& operator[](int position)
```

returns a reference to element whose position is position.

Accessing the State of a Sparse Vector

The methods

```
int get_dim()
int num_elements()
```

return the size of the vector and the maximum number of non-zero elements that can be represented, respectively. The method

```
int is_full()
```

returns TRUE if the maximum number of non-zero elements has been reached. It returns FALSE otherwise. The methods

```
int get_status()
int set_status(int stat)
```

return or set the status of the vector. These methods are analogous to the corresponding methods for the class DD_floatarray. The methods

```
int get_first_index()
int get_last_index()
```

return the index of the first and last non-zero elements, respectively.

Sparse Vector Manipulation Methods

The method

```
int add_element(int position, float value)
```

inserts the element value in position position within the vector, overwriting any previous value at that position. It returns SUCCESS or FAILURE. The method

```
int delete_element(int position)
```

removes the element at position `position` from the vector, returning `SUCCESS` if the element is removed, `FAILURE` if the element at `position` is zero.

The method

```
sp_sparsevector* overlay(sp_sparsevector* vector)
```

creates a new vector with the dimension of the source and the number of elements equal to the sum of those for the source and the argument. The elements in the argument are inserted into the source vector in the proper order. If the two vectors have an element at the same position, the value is updated to the value of the argument vector. In the event that the number of elements has been exceeded, the vector contains the first `num_elements` elements from each of the vectors. The method

```
void print()
```

prints the elements of the vector and their corresponding indices to standard output.

Mathematical Functions

The operators

```
void operator += (float scalar)
void operator -= (float scalar)
void operator *= (float scalar)
```

perform scalar addition, subtraction, and multiplication on the values of the vector, respectively. The method

```
sp_sparsevector *add(sp_sparsevector *vector)
```

adds the vector `vector` to the source. The sum is returned as a new vector with the maximum number of non-zero elements set to the lesser of the sum of the `num_elements` of the arguments or the dimension. In the event that the dimensions of the two vectors are not equal, a vector of dimension 0 and status -1 is returned. The method

```
float dot_product(sp_sparsevector *vector)
```

calculates the dot product of the two vectors and returns the result. In the event that the vectors are not of the same dimension, a value of zero is returned. An equivalent method exists for the class `dp_sparsevector`.

8.4 File Interface for Array and Vector Classes

The `DD_floatarray_file` class implements a file interface for two dimensional single precision floating point arrays. A file consists of a header and the array elements stored in row-major order. The file header contains a magic number, the file type, the size of the data elements (in bytes), the number of rows and columns of the array, and the array elements. The magic number is a means of ensuring that the file version is consistent with the library functions. The file type is a means of ensuring that the type of structure is consistent with the library functions. Currently, only files of two

dimensional arrays are supported. The state of a file consists of its status, the mode in which it was opened, and the number of rows and columns contained in the file. The status of a file can any one of the following:

OPEN
CLOSED
MAGIC_NUM_ERROR
OPEN_ERROR
HEADER_READ_ERROR
HEADER_WRITE_ERROR
FILE_TYPE_HEADER_ERROR, and
BAD_HEADER

Each of these status codes are discussed below. The modes in which a file can be opened are: READ, READ_AND_WRITE, and WRITE. These modes are also discussed in detail below.

When using these classes, it is important that the file permissions allow for at least Mentat group access since Mentat applications actually use the "mentat" user id instead of the user's user id. As with the other classes presented, analogous methods exist for the classes DD_chararray_file, DD_doublearray_file, and DD_intarray_file.

File Initialization Methods

The constructor

```
DD_floatarray_file(string *fname, int mode)
```

creates and opens the file, where file_name is the name of the file and mode is the mode in which the file is to be opened. The allowable modes are: READ, READ_AND_WRITE, and WRITE. A file opened for READ allows the file to be read. A file opened for WRITE allows the file to be written to. However, if a file is opened for WRITE, any existing file will be erased. This is the only mode in which a file will be created. A file opened for READ_AND_WRITE may be either read from or written to. However, the file must already exist. If the mode is READ or READ_AND_WRITE, the header information is read from the file and the file is ready to be manipulated. If the mode is WRITE, then the header information must be explicitly written using a call to file_alloc() (described below). The status is set to OPEN if the open command is successful. In the event that the open command is not successful, the status is set to one of the following error codes: OPEN_ERROR, indicating that the file does not exist, the file permissions have not been properly set, or the disk is full, MAGIC_NUM_ERROR, indicates that the version number contained in the file is inconsistent with the version number expected by the class, FILE_TYPE_HEADER_ERROR, indicating that the file type for is inconsistent, or HEADER_READ_ERROR or HEADER_WRITE_ERROR, indicating that there was an error while reading or writing the header information (such as the file is malformed). Further, an open command may fail if the system-defined limit for number of file pointers allowed for one file is exceeded. SUCCESS is returned upon successful completion, FAILURE otherwise.

The methods

```
int file_alloc(DD_floatarray *data)
int file_alloc(int num_row, int num_col)
```

prepare a file that has been opened for write to accept data. The parameter `data` is a `DD_floatarray` from which the number of rows and columns can be extracted. Writes the header information to the file.

They return `SUCCESS` if the header is successfully written. In the event that the header information is not written, `FAILURE` is returned, and the status of the file is set to `HEADER_WRITE_ERROR`.

This function only prepares the file for writing, in order to write to the file, a call to one of the write methods described below must be made. The file must already have been opened, and its mode must be `WRITE`.

The method

```
int close_file()
```

closes the file, resets the file status to `CLOSED`, and returns `SUCCESS` or `FAILURE`, depending on the outcome of the close.

Accessing the State of a File

The methods

```
int get_row_size()
int get_col_size()
int get_data_type_size()
int get_header_size()
```

return the various attributes of the file. The first two methods return the number of rows and columns contained within the source file, respectively. The latter two methods return the size of the data to be stored in the file, and the size of header within the file, respectively.

Reading from a File

The method

```
DD_floatarray* read_row(int row)
```

reads the row specified by the parameter `row` from the file, places it in a $1 \times$ `num_columns` `DD_floatarray` and returns the array. The call returns `SUCCESS` if read is successful, otherwise it returns an empty array whose status is -1. The parameter `row` must be a valid row within the range specified in the header of the file.

The method

```
DD_floatarray* read_row(int row, int start_col,
                        int end_col)
```

reads the row specified by the parameter row from the file, places it in a $1 \times$ (end_col−start_col) DD_floatarray and returns the array. It returns SUCCESS if the read operation is successful, or an empty array whose status is -1, otherwise. The parameter row must be a valid row within the range specified in the header of the file. The parameters start_col and end_col must also be within the appropriate ranges.

The method

```
DD_floatarray* read_col(int column_number)
```

reads the column specified by column_number into an appropriately sized DD_floatarray. The array is then returned. In the event of an error, an empty array whose status is -1 is returned. The column_number must be within valid ranges as specified in the file header.

The method

```
DD_floatarray* read_region(int ul_row, int ul_col,
                           int lr_row, int lr_col)
```

reads the portion of the array bounded by [ul_row, ul_col], [lr_row, lr_col], creates and places it a new DD_floatarray, and returns the result. In the event of a read error, an empty array whose status is -1 is returned.The region must be a valid region within the range specified in the header of the file.

The method

```
DD_floatarray* read_array()
```

reads the entire array from a file, creates and places it in a new DD_floatarray and returns the result. In the event of a read error, an empty array, whose status is -1, is returned.

Writing to a File

The methods

```
int write_row(DD_floatarray* data, int row)
int write_row(DD_floatarray* data, int row,
              int start_col, int end_col)
```

write row information to the file. In the first method, row row from the DD_floatarray data is written to the corresponding position in the file and a SUCCESS or FAILURE is returned depending on the outcome of the write. The value of row must be a valid row within the range specified in the header of the file. The second method writes row row between columns start_col and end_col from the DD_floatarray data to the corresponding position in the file. It returns SUCCESS or FAILURE depending on the outcome of the write. The values of row, start_col, and end_col must be valid within the ranges specified in the header of the file.

The method

```
int write_col(DD_floatarray* data, int column_number)
```

writes column `column_number` of `data` to the corresponding position in the file. It returns `SUCCESS` or `FAILURE` depending on the outcome of the write. The value of `column_number` must be within valid array bounds, and the number of rows in the array must equal those specified in the header.

The method

```
int write_region(DD_floatarray* data, int ul_row,
                 int ul_col, int lr_row, int lr_col)
```

writes the region of the `DD_floatarray data` bound by [`ul_row`, `ul_col`], [`lr_row`, `lr_col`] to the corresponding position in the file. It returns `SUCCESS` or `FAILURE` depending on the outcome of the write.

The method

```
int write_array(DD_floatarray* data)
```

writes the entire `DD_floatarray data`, to the file, and returns `SUCCESS` or `FAILURE` depending on the outcome of the write. The number of rows and columns in data must match those specified in the file header.

The method

```
void print(string *file_name)
```

opens the file specified by `file_name`, prints the contents in row-major order, and closes the file.

9.0 Problems at Run-Time

A number of run-time errors can occur while running a Mentat application. Some of the errors are caused by confusing the C++ semantics with the semantics of MPL. After gaining experience with the system, the Mentat programmer is less prone to make some of the mistakes that lead to run-time errors.

Future stack overflow and underflow

If you fail to execute an `rtf()` for every Mentat member function invoked, then you will eventually get the error “FUTURE STACK OVERFLOW.” For example,

```
yy_matrix* mop::f_tsp(string* m) {
   yy_matrix* mtp = in_ftsp(m);
   rtf(mtp); // if you omit this
}
```

will eventually cause an overflow.

An `rtf()` is required for void Mentat member functions as well. On the other hand, if you include more “rtf’s” than the number of functions called, you will get the error “FUTURE STACK UNDERFLOW.” This error commonly occurs if private members

on Mentat classes are defined to be helper functions, but the user has changed the `return()` to an `rtf()`. In this case, only a return is necessary.

Memory problems

If you run out of virtual memory on your machine, you may see the error "OUT OF MEMORY IN RESOLVE, GOODBYE" or "out of memory in message constructor." Both of these messages indicate situations that are eventually fatal and are most likely the result of a memory leak in your program. This may be caused by not releasing the heap allocated results of Mentat member functions that return pointers.

Failure to instantiate an object

If the scheduler cannot locate a class binary in the current directory or in $MENTAT_USR_BIN or the binary does not have group execute permission, then the scheduler may not be able to create the object and a error message will be issued. When the message is reported, the user needs to check which of these situations applies and take the appropriate action.

Core dumps and memory faults

There are several possibilities. Check if Mentat is running by issuing the **list_objects** command. This will tell you if the **im**s are still alive. You can also see if the **tmu**s are still working by running the dotest_fib application in $MENTAT/sources/examples/fib. If this checks out, then there is most likely a bug in the user's code. There are two cases: (1) bad MPL code, and (2) incorrectly generated C++ code. We recommend the following debugging strategy: use the Mentat I/O system (`mstream`) to help locate the crashing member function. If it is the user bug, then locating the context will hopefully lead to an obvious bug fix. If the user is convinced that the MPL code looks fine, then the *file.trans.c* file must be inspected. Search for the name of the member function to find the analogous C++ member function. If the C++ member function seems to have been generated incorrectly, the user has two options: if the user is a solid C++ programmer the person can attempt to modify the C++ code and recompile from the *file.trans.c* file on down. If not, the user should consult the MPLC section of this manual and find out the cases for which the MPL compiler may generate bad code. If the user cannot match their bug to any known compiler bug described in this section, then we wish to be informed of this situation. Often, a core dump or memory fault is the result of an undefined variable.

If Mentat is not running, then the Mentat run-time system has crashed. Fortunately this is quite rare. If the *config.db* files get damaged, the **im** may crash. If this happens, kill Mentat (with **kill_mentat**), fix the *config.db* file, and try again. If the *config.db* file is not damaged, then please let us know about this.

Mentat application hangs

There are two cases where <class>.`destroy()` may cause a run-time hang-up: (1) the sequence `create()`, `destroy()`, `create()` on the same Mentat class object, and

(2) the sequence `destroy()`, `destroy()` on the same Mentat class object. The system will catch these most of the time and will either print a message or just "ignore" the spurious `destroy()` or `create()`, but there may still be cases where (1) or (2) causes a hang-up. Note that case (1) actually makes sense and we plan on supporting this in the near future.