Mentat User's Manual

Andrew S. Grimshaw
Edmond C. Loyot, Jr.
Sherry Smoot
Jon B. Weissman

Computer Science Report No. TR-91-31
November 3, 1991

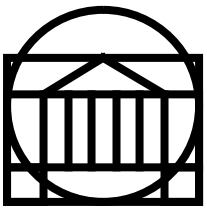
Abstract

Mentat User's Manual

Andrew S. Grimshaw
Edmond C. Loyot, Jr.
Sherry Smoot
Jon B. Weissman

grimshaw@virginia.edu
ecl2v@virginia.edu

November 3, 1991



DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
THORNTON HALL
CHARLOTTESVILLE, VIRGINIA 22903-2442
(804) 982-2200 FAX: (804) 982-2214

This work was partially supported by NASA grant NAG-1-1181.

Mentat User's Manaul

1.0 Introduction

The Mentat system is made up of three major components: the Mentat Programming Language compiler (MPLC), the Mentat run-time system (RTS) and a run-time monitor tool called MentatView. These components together make up a complete parallel and distributed programming environment that is both easy to use and architecture independent. MPLC is a compiler for the Mentat Programming Language(MPL)[1]. It translates MPL into C++ with embedded Mentat RTS calls. These calls to the RTS automatically manage all communication and synchronization. The C++ code with embedded RTS calls is then compiled with a standard C++ compiler, producing executables that can be run by the Mentat RTS. The RTS is a distributed system for executing parallel programs. It is based on the Macro-Data-Flow model. MentatView is a program that allows a user to monitor the Mentat RTS as it executes a program. It displays various performance information about the system. This document explains in detail how to use each of these components. Chapter 2 discusses how to use MPLC. Chapter 3 explains how to install and operate the Mentat RTS. Chapter 4 discusses how to use MentatView. Several chapters also contain a short section listing any known problems or bugs relating to the Mentat component described in that section. These sections are marked with a shaded box and the Mentat user/developer is encouraged to read these sections carefully.

Please note that this document is an alpha release, as is the current version of the Mentat system. As such, it is likely to have mistakes and deficiencies. If you find any mistakes or have any comments, suggestions or questions, please do not hesitate to call or email the authors.

1. For details on the MPL see the “Mentat Programming Language (MPL) Reference Manual”. It is available with the standard Mentat distribution. You should read the MPL reference manual before attempting to write applications using Mentat.

2.0 Mentat Programming Language Compiler (MPLC)

2.1.0 Introduction to MPLC

The Mentat Programming Language (MPL) is an object-oriented, parallel programming language. It is designed to support high degrees of parallelism, yet still be easy to use. MPL supports high degrees of parallelism because it uses the Macro Data-Flow model [Liu86] as its underlying model of computation. It is easy to use because the compiler automatically manages all communication and synchronization. Since MPL syntax is a superset of the object-oriented programming language C++, it simplifies construction of large parallel programs and encourages code reuse.

2.2.0 How to Use MPLC

MPL programs are compiled by the Mentat Programming Language compiler (MPLC). MPLC consists of a shell script named “mplc” and a program named “mplfront”. “Mplc” parses the command line, determining the input file and any command line switches. It then invokes the “cpp” preprocessor on the input and pipes the result to “mplfront”. “Mplfront” parses the expanded code and generates code to perform runtime data flow detection. It then generates C++ code with embedded calls to the Mentat runtime system. The embedded calls generated by “mplfront” handle all communication and synchronization in the program.

MPLC is invoked from Unix by running the shell script “mplc”. “Mplc” accepts several command line switches and a single argument. The switches accepted by “mplc” are described in the next section. The argument that “mplc” accepts is the file name of the file to be compiled. It must be a file containing a valid MPL program and the file name must end in “.c”. “Mplc” outputs a file containing C++ code with embedded calls to the Mentat runtime system. The output file name is created from the input file name by inserting “.trans” before the “.c”. For example, if the input file name is “test_matrix.c” the C++ output file name will be “test_matrix.trans.c”. The output file must then be compiled with a C++ compiler and linked to the appropriate Mentat system object files to produce an executable program (see Figure 1 below).

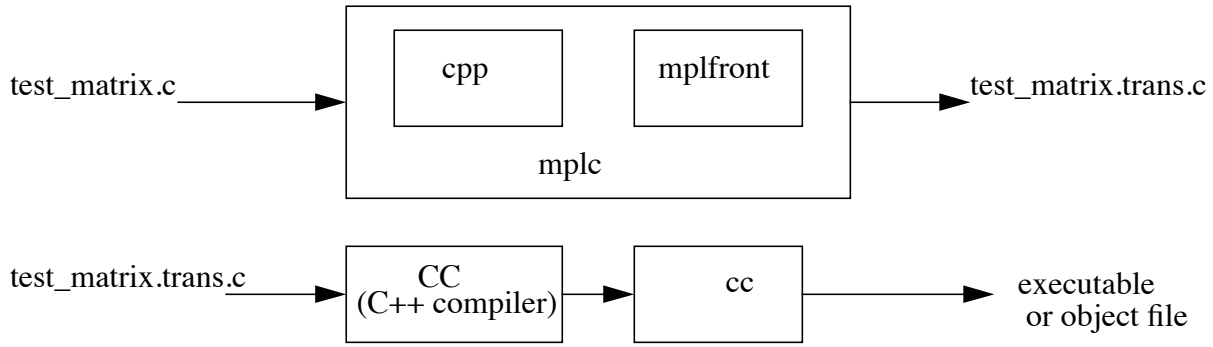


Figure 1. MPL Translation steps.

A typical makefile for compiling Mentat objects is shown below. See also the examples in

the examples directory provided with the distribution. There are several points to note about the structure of this makefile. Notice that there is a separate compilation entry for each Mentat class (e.g. test_matrix, queue) and the main program (if a main was specified). This is because each Mentat class will have a unique executable associated with it. The name of the executable MUST be identical to the Mentat class name. For this reason, the programmer must make sure that there is a separate file for each Mentat class definition. The makefile follows the logical steps illustrated in Figure 1. In order to build the Mentat class executables (e.g. test_matrix), the makefile does the following: it first checks the dependency list (i.e. $(HEADERS), matrix_class.h, ...) to see if any of these files are newer that test_matrix, if so, test_matrix is out of date and must be rebuilt. To do this, the makefile will execute the specified commands in the action list following the target name:

- MPLC is run on the Mentat class source file (e.g. test_matrix.c) producing C++ file
- The C++ compiler is run on the resulting C++ file (e.g. test_matrix.trans.c)
- Implicitly the C compiler will then be called to produce the executable
- The executable is copied to $MENTAT_BIN so Mentat will be able to locate it (see Section "Searching for Configuration and Executable Files")

```
MPLC = mplc $(C++) $(CFLAGS) test_matrix.trans.c -o test_matrix \
MPLC = mplc
C++ = CC
CFLAGS = -I/usr/include/CC -I/users/mentat/h -Dsun_mess -DC++ \
 -DBSD -DTIMES -DREG=register $(C++)
LFLAGS = -L/users/mentat/lib -lmentat -lC -lm -Bstatic

test_matrix: $(HEADERS) matrix_class.h test_matrix.c $(OBJECTS)
 $(MPLC) $(CFLAGS) test_matrix.c
 $(C++) $(CFLAGS) test_matrix.trans.c -o test_matrix \
 $(OBJECTS) $(LFLAGS)
 cp test_matrix $(MENTAT_BIN)
 rm test_matrix.trans.c test_matrix.trans.o

matrix_class: $(HEADERS) matrix_class.h matrix_class.c $(OBJECTS)
 $(MPLC) $(CFLAGS) matrix_class.c
 $(C++) $(CFLAGS) matrix_class.trans.c -o matrix_class\
 $(OBJECTS) $(LFLAGS)
 cp matrix_class $(MENTAT_BIN)
 rm matrix_class.trans.c matrix_class.trans.o
```

Figure 2. A Typical makefile

2.3.0 Command Line Switches

"Mplc" recognizes the following command line switches. All others are ignored:

-E Run only the "cpp" preprocessor on the input file. The result is directed to standard output.

-N	Do not do any Mentat code generation. When the -N option is selected the output contains no Mentat runtime system calls and no Mentat server code. Syntax is checked, but little else.
-I*pathname*	Add *pathname* to the list of directories that the "cpp" preprocessor will search for #include files.
-D*name*[=*def*]	Define a symbol *name* to the "cpp" preprocessor. This is equivalent to a #define directive in the source code. If no *def* is given, *name* is defined as '1'.

2.4.0 MPLC Bugs

This is an alpha release of MPLC. It contains bugs both known and undiscovered. There are several categories of bugs, and each is discussed below. Included in the discussion are some suggestions for getting around these bugs. Please report newly discovered bugs to the author via email so that they may be fixed in future versions. The User's Guide contains a comprehensive error Appendix at the end of the document.

2.4.1 Parse Errors

If "mplfront" quits with the message "*program_name*: line *line_number*: syntax error!", then it has encountered a syntactic construct that it could not parse. Check to make sure the construct is valid MPL code. If it still fails to parse, try using a different syntax. If the problem involves a statement or declaration with a type name, make sure the type is defined and its syntax is correct. The problem could be that "mplfront" is mistaking a type name for a variable name.

2.4.2 "Mplfront" Core Dumps

These are the most difficult bugs to deal with. First, the code that is causing the core dump must be isolated. I usually do this by commenting out parts of the code until I've determined which code fragment is causing the problem. Try substituting an alternative syntax. If you are using a very complex or archaic syntax try using a simpler or more modern style.

2.4.3 Errors when Compiling the ".trans.c" File

These errors occur when "mplfront" generates invalid C++ code. The work around is to edit the ".trans.c" file and fix the incorrectly generated code. Unfortunately this must be redone with each compile. However, bugs of this nature are very rare.

2.4.4 Known Bugs

The following are the known bugs in version 0.1 of "mplfront".

• The Annotated C++ Reference Manual [Ellis90] states that in-line member functions declared inside of a class are not type checked until the complete class declaration has been seen (p. 171). Mplfront currently does type checking as the in-line member function is parsed. This often causes spurious errors, particularly with some of the standard header files, like streams.h. To correct this problem mplfront must be rewritten to do type check-

ing as a separate pass, after parsing. The work around is to write the problem member function outside of the class using the in-line keyword.

• Conversion functions [Ellis90 p. 272] cause a syntax error. Mplfront is built around a public domain C++ grammar [Roskind90]. 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 will be released late in the summer of 1991. This new version will be incorporated into future versions of mplfront. There is currently no work around for 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 ifdef'd.

2.5.0 Unsupported C++ Features

The following features of C++ are not supported.

• As per the [Ellis90, 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, as per [Ellis90], are not supported.

• 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 [Ellis90]. This may cause parse errors when compiling the standard header files. The offending code must be modified or commented out.

2.6.0 Unsupported MPL Language Features

The following features of MPL have not yet been implemented:

• Guarded statements. A default select/accept is generated for each Mentat object. However, users cannot define their own select/accepts.

• Create() cannot be overloaded by the user.

• Location hints. Although these are not currently implemented, the effect of the location hint *co-locate* can be achieved in the following way. There are actually two create() functions defined. The first is create() (with no arguments). This 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.

• Overloading of Mentat object member functions is not supported. The work around is to uniquely name each member function.

• Inheritance using Mentat classes is not supported.

2.7.0 Bug Fixes Pending

• Derived Mentat classes do not work.

• 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 fixed-sized argument, even if class has

size_of () member defined.

• Assigning Mentat class variables to Mentat objects returned from a Mentat class member function (by ptr) will not resolve properly, and the assignment will not be performed. To work around this, the Mentat object must be assigned first to a temporary variable and then the Mentat class variable should be assigned to the temporary.

• Default arguments to Mentat member functions are disallowed.

2.8.0 How to Get Help

If you are having difficulty using the compiler or if you have found a bug please contact Ed Loyot (ecl2v@virginia.edu) or Andrew Grimshaw (grimshaw@virginia.edu). Please include a detailed description of the problem you are having, and include a copy of the offending code and the ".trans.c" file (if any). A daytime phone number would also be helpful. Any comments or suggestions will be appreciated.

References

[Ellis90] Margaret A. Ellis and Bjarne Stroustrup, "The Annotated C++ Reference Manual", Addison-Wesley Publishing Company, 1990.

[Liu86] J. W. S. Liu and A. S. Grimshaw, "An Object-Oriented Macro Data Flow Architecture," *Proceedings of the 1986 National Communications Forum*, September, 1986.

[Roskind90] The public domain C++ grammar is copyright 1989, 1990 by James A. Roskind (jar@florida.HQ.Ileaf.COM).

3.0 Mentat Run-Time System (RTS)

3.1.0 Introduction to the RTS

This section describes the operation of the Mentat run-time system (RTS). It does not describe the internal functions of the RTS. The section is divided into seven short sections. Particular attention should be made to Section 3.2, Installation, and Section 3.3, Configuration.

3.2.0 Installation

This section covers setting up directories, unpacking the tar file, system administration issues, and a brief tutorial and run-through to ensure that the installation has been done correctly.

1) Decide on a name for the directory in which Mentat will be installed (for example, "/users/mentat"). This pathname should be exported as the environment variable MENTAT when actually using the RTS.

2) Have your system administrator create a user **mentat** and add **mentat** to the **kmem** group. This is necessary in order for the Mentat RTS to extract from the operating system load information used in making scheduling decisions. The home directory for **mentat** should be the same as the pathname to be assigned to the MENTAT environment variable.

3) We recommend that you also create a group **mentat** to which users of Mentat belong. This simplifies protection.

4) Login as **mentat** and assign a password to the account. If your machine uses NIS ("Yellow Pages"), you will need to run *yppasswd*; if not, you will need to run *passwd*. Ask your system administrator for advice if you are not sure how password information should be modified on your system.

5) Set your umask to 022. Unarchive the distribution file. This file (mentat.tar.Z.crypt) is an encrypted, compressed tar archive. Run the following to extract the distribution:

```
crypt ??? < mentat.tar.Z.crypt | uncompress | tar xpBf -
```

where "???" is the encryption password you have been provided. If you do not have a password, send e-mail to the authors which includes your e-mail address, name, and US Mail address to request a password.

6) Set the environment for Mentat by running one of the following examples (first changing `/users/mentat` to your own choice for the Mentat home directory). If your login shell is the Bourne shell or a derivative (**sh** or **ksh**), run the following and also add the lines to the .profile start-up file of each Mentat user with a Bourne/Korn login shell:

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

If your login shell is the C shell or a derivative (**csh** or **tcsh**) run the following and also add the lines to the .login start-up file of each Mentat user with a C shell login shell:

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

These commands tell the system where to find configuration files and the executables appropriate to your architecture. Every user of Mentat must run these commands.

7) Run *set_protection*. This sets all of the setuid bits and other protection bits.

8) Set up a configuration file for your site. We recommend using only a few (3-4) hosts until you have gained experience with Mentat. See Section 3.3 on setting up your configuration.

9) Run *setdefaultconfig*.

--------- The following section executes and tests the installation -----

10) Verify that you can "rsh" to each host in the configuration. Some sites "trust" local machines. If your site does not, you will need an appropriate .rhosts file. See your system administrator for details.

11) Run *start_mentat*.

12) Run *dotest*.

13) Run *mcp*.

14) When the prompt appears, type "shutdown<cr>". The program should print

```
killing im#0
killing im#1
....
system_prompt>
```

If the program hangs and does not display the system prompt, then type <ctrl C>. This should give you the system prompt. Then "su" to mentat and execute "kill_mentat".

If the program did not hang then everything is fine. You are ready to begin using Mentat.

3.3.0 Configuration - The config File

The "config" file specifies the hosts that are to be used by Mentat. The "config" file is loaded by all Mentat objects, including the scheduler ("h_im"), during initialization. It provides the names of the hosts in the system, as well as the number of hosts. A sample config file is shown

below.

```
//*********************************************
// Sample Mentat configuration file for Sun workstations
// ----------------------------------------
//*********************************************
// First the number of stations
4
// Then the names of the stations
//****** 60s ******
hopper
batik
calico
cassandra
// Comments may appear on any line that has // in the first two positions.
// This is a comment.
//horus
```

The file consists of three types of entries, comments, the number of stations, and the names of the stations. A comment line is any line that begins with "//". The first non-comment line is assumed to be an integer that represents the number of stations. If the number is larger than the actual number of stations an error will occur. After the number of stations come the station names, e.g.,"hopper". The names must be recognizable by the local name server. At some sites the full name will be required, e.g., hopper.cs.virginia.edu. Care must be taken not to misspell station names. We have found that typing the names in once, and then commenting them as desired works very well, e.g., "//horus". This allows you to use different subsets of the stations at different times. A control application is being developed that will allow the user to graphically manipulate the configuration.

3.3.1 "Setdefaultconfig"

"Setdefaultconfig" is a "csh" script that sets up a default "threshold_config" using the "config" file as a basis. It uses the program "numnodes". "Numnodes" returns the number of hosts specified in the "config" file. "Setdefaultconfig" then creates a "threshold_config" file configured for a single subnetwork, one Mentat object per host, a transfer limit of (numnodes-1), and a location policy of "best-most-recently". The "threshold_config" file can be edited by hand. If you are not using "csh", type "csh /users/mentat/bin/setdefaultconfig".

3.4.0 Configuring the Mentat Scheduler

(may be skipped if using setdefaultconfig)

One of the features of the Mentat scheduler is that it can be parameterized via configuration files. This is achieved by running a configuration program before the Mentat RTS is started. The configuration program creates a file with the parameters needed by the scheduler in making its decisions.

The configuration program, “vconfig”, serves two purposes (see Figure 3). First, “vconfig” creates a file (“threshold_config”) which contains the parameters needed by the scheduler. Second, “vconfig” is used to display the current scheduler configuration whenever a “threshold_config” file exists. The file “config” indicates to the instantiation managers the number of nodes comprising the system and their names/addresses. “Config” is only used in the Sun workstation environment.

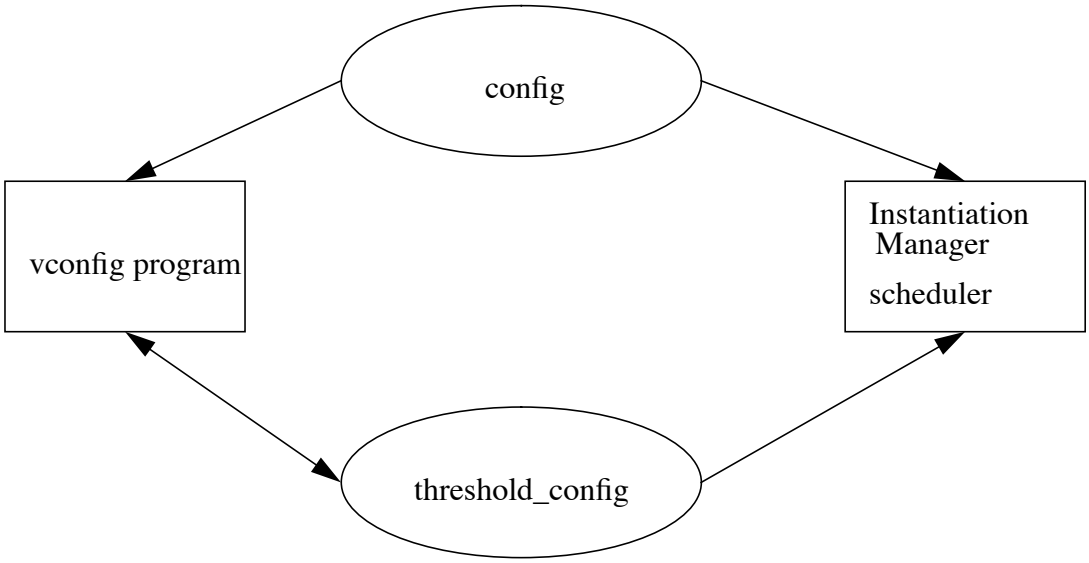


Figure 3. Scheduler Input Files

When “vconfig” is executed for the first time it creates a new “threshold_config” file. The scheduler parameters are requested in the following sequence:

```
----------------------- MENTAT-RTS -----------------------
+ This program changes the input threshold values in the +
+ threshold_config file which are used by the MENTAT     +
+ Instantiation Manager when making scheduling decisions.+
-----------------------------------------------------------

 Enter the threshold parameter for local decision
 q (queue-length), m (free-memory (Kb)), c (cpu idle (%)) : q
 Enter the threshold value = 1
```

The valid parameters are “q”, “m” and “c”. The option “q” corresponds to the running queue-length of the underlying system which might include Mentat objects. The option “m” corresponds to the available free memory on the local host (in Kbytes). The option “c” corresponds to the percentage of idle cpu. The threshold transfer policy is set by selecting any of these parameters along with a threshold value. If we want the scheduler to accept Mentat objects only if the running queue of the underlying system has less than two processes, we must indicate “q” as the parameter and “1” as the corresponding threshold value. The values “q” and “1” are recommended for general use.

The transfer policy parameters depend on the working environment. In the hypercube environment, the transfer decision is based solely on the number of Mentat objects on the node. In the Sun workstation environment, the number of Mentat objects is used for controlling the initial arrangement of Mentat objects in the system. The next query in the configuration program is:

```
Enter the maximum number of MENTAT objects per node = 1
```

The value selected for this parameter indicates the number of Mentat objects allowed on a host. If the number of Mentat objects on a host is less than this value, the transfer policy is to accept new objects. Otherwise, the object is sent to another node. The recommended value for this parameter is 1.

Logical sub-networks are created by grouping homogenous nodes of a system. The instantiation managers must be told the number of logical sub-networks and their respective nodes. The "config" file provides the nodes information on the entire system and "vconfig" divides the entire system into logical sub-networks. On the hypercube there can only be one logical sub-network.

```
Enter the number of logical sub-networks = 1
Enter transfer limit between sub-networks = 0
```

The number of logical sub-networks indicates into how many sub-networks the entire system is divided. If we have twenty workstations available for the Mentat system, we may create two or more sub-networks representing the physical locations of the workstations. If we select "0" as the value for the transfer limit, we configure two or more independent systems. If we select a value greater than "0", it must be less than the number of sub-networks. One sub-network is the recommended value.

Each logical sub-network may have a different number of nodes. Also, the internal transfer limits may be different for each sub-network. For each sub-network, these parameters must be specified:

```
Enter the number of nodes in sub-network 1 = 14
Enter the transfer limit for sub-network 1 = 6
Enter the number of nodes in sub-network 2 = 6
Enter the transfer limit for sub-network 2 = 4
```

The number of nodes indicates how many nodes there are in the sub-network. The transfer limit represents the internal transfer limit between nodes of the sub-network. The transfer limit refers to the maximum number of nodes that can be visited in scheduling a particular Mentat object. The transfer limit must always be less than the number of nodes.

The location policy algorithm to be used by the scheduler must also be indicated. Currently, there is a choice between three distinct algorithms:

```
Enter strategy (0-Random, 1-Round Robin, 2-B_M_R) = 2
```

"0" is a random algorithm, "1" is a round robin algorithm and "2" is the best-most-recently algorithm. "2" is the recommended algorithm because it tends to perform best under a wide variety of circumstances[1]. The program "setdefaultconfig" reads the current file and sets the values mentioned above to default values.

Figure 4 shows a "vconfig" run. The configuration selected divides a system of 14 nodes

into three independent logical sub-networks. In this case, the sub-networks are used to group homogeneous processors.

```
Unix$ vconfig
      ------------------------ MENTAT-RTS ------------------------
      + This program changes the input threshold values in the    +
      + threshold_config file which are used by the MENTAT        +
      + Instantiation Manager when making scheduling decisions.   +
      ------------------------------------------------------------
      Enter the threshold parameter for local decision
      q (queue-length), m (free-memory (Kb)), c (cpu idle (%)): q
      Enter the threshold value = 1
      Enter the maximum number of MENTAT objects per node = 1
      Enter the number of logical sub-networks = 3
      Enter the transfer limit between sub-networks = 0
             Enter number of nodes in sub-network 1 = 2
             Enter transfer limit for sub-network 1 = 1
             Enter number of nodes in sub-network 2 = 10
             Enter transfer limit for sub-network 2 = 6
             Enter number of nodes in sub-network 3 = 2
             Enter transfer limit for sub-network 3 = 0
      Enter strategy (0-Random, 1-Round Robin, 2-B_M_R) = 2
```

Figure 4. Configuring the Scheduler for the Sun Workstation Environment.

In Figure 5.,“vconfig” is used to display an existing configuration. This configuration is read from the files “config” and “threshold_config”. The number of nodes of the entire system is

1. Grimshaw, A. S., and Virgilio 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.

shown along with the processors comprising the different sub-networks.

```
Unix$ vconfig
      +++++++++++++++++++++ MENTAT-RTS +++++++++++++++++++++
      + Current set-up for the scheduler+
      +                                                     +
      + threshold parameter: q                              +
      + threshold value: 1                                  +
      + maximum number of MENTAT objects per node : 1       +
      + number of logical sub-networks : 3                  +
      + transfer limit between sub-networks : 0             +
      + location strategy : 2                               +
      +++++++++++++++++++++++++++++++++++++++++++++++++++++++
      Do you wish to see the network set-up (y or n)? y

              ***********************
              logical sub-network 1
              number of nodes = 2
              transfer limit = 1
                    checkered
                    madras
              ***********************
              logical sub-network 2
              number of nodes = 10
              transfer limit = 6
                    abraxas
                    antares
                    harlequin
                    hopper
                    horus
                    mithras
                    neptune
                    phoebus
                    polka-dot
                    surya
              ***********************
              logical sub-network 3
              number of nodes = 2
              transfer limit = 0
                    chaos
                    shamash

      Do you wish to change the set-up (y or n)? n
```

Figure 5. Displaying the Scheduler Configuration.

3.5.0 Searching for Configuration and Executable Files

Mentat will perform limited searching for files. Mentat will search first in the directory in which "start_mentat" was executed, and then in "$MENTAT_BIN" (which is set equal to the appropriate subdirectory in the $MENTAT directory by either $MENTAT/env_sh or $MENTAT_env_csh). The exceptions to this rule are programs that are executed directly from the Unix command line, e.g., mcp, mentatview, start_mentat, setdefaultconfig, vconfig, and kill_mentat. Those programs will search first in the current working directory and then in "$MENTAT_BIN"

3.6.0 What the Mentat User Needs to Know

Once Mentat is properly configured, there are a number of simple steps that Mentat users must perform or at least be aware of before running MPL programs. The users environment must be configured in the same way as the "user **mentat**" (see Section Installation). The user will the set up the environment as described in paragraph 6) by adding the appropriate lines to their .login file. The interested user can type set (sh/ksh) or setenv (csh/tcsh) to see what Mentat environment variables were defined. The user must be aware of the rules that Mentat employs for locating executables (i.e. class object instances) as described in the previous section. We recommend that the user create a working directory to keep Mentat class executables and Mentat config files. If the user brings up Mentat, then it should be launched from this directory. If Mentat is already running (started by someone else), then the user must assume that Mentat will expect the class executables to be in $MENTAT_BIN. The user is encouraged to look in $MENTAT_BIN to see what is there, especially if the MPL program did not work as expected.

A related problem that the user must be aware of is file permissions on executables. During program execution, the Mentat RTS will need execute permission for each Mentat class executable that is part of the user program. If the system administrator has set up a "**mentat**" group which contains the user and "user **mentat**", then the user must insure that the group has execute permission on the executables. Otherwise, the user must insure that all others have execute permission. Wrongly set permission bits is another common source of unexpected program failure. For other runtime errors, consult Section "Miscellaneous Problems and Errors".

3.7.0 Mentat Control Program (mcp)

"Mcp" is a very simple command interpreter that accepts three types of commands. When invoked you are presented with the following prompt

mcp command >>

If you type "**quit**<cr>" then mcp exits.

If you type "**shutdown**<cr>" mcp sends a shutdown message to the instantiation managers ("h_im") on each of the hosts named in the config file. It then terminates. If one of the instantiation managers does not respond, then the "mcp" will hang trying to talk to that manager. The only recourse at this point is to use "kill_mentat".

The instantiation manager can also be used to instantiate Mentat objects. The syntax is:

mcp command >> **mentat_class_name arguments**

An instance of mentat_class_name is instantiated, and the string following the class name is sent to the first member function of the instantiated object. In order to use this capability the class must be defined as follows[2]:

```
regular mentat class example
{
 public:
      int main_loop(string* arg);
      .... possibly other member functions
};
```

It is expected that in the future most Mentat programs will be invoked from the Unix command line and not from "mcp".

3.8.0 Miscellaneous

3.8.1 "Start_mentat"

"Start_mentat" uses the "config" file to determine the number of hosts and their names. It then invokes a remote shell on each of the hosts. The remote shell consists of a command to change the directory to the current working directory, and then to fire up an instantiation manager ("h_im").

"Start_mentat" is the simplest way to start Mentat running.

If Mentat is already running on a particular host the error, "Recv Socket BIND : Address already in use" will appear. Before continuing you must first execute "kill_mentat", and then restart Mentat.

Keep in mind that the stderr and stdout of all remote processes is the window in which start_mentat is executed. Because all Mentat objects are the child of one of the remote shells, they all share the same stdout. Separate input/output streams can be realized using the **mstreams** facility, see Chapter 5.

3.8.2 "Shutdown_mentat"

"Shutdown_mentat" uses the "config file" to determine which hosts Mentat is running on. It then invokes a remote shell on each of the hosts to kill the Mentat processes running on that host.

3.8.3 "Kill_mentat"

"Kill_mentat" uses the "config file" to determine the number of hosts and their names. It then invokes a remote shell on each of the hosts that executes "kill -9 -1", killing all processes with the user id of the person who invoked it. To kill all Mentat processes, first "su mentat" to change your effective user id. Then invoke "kill_mentat".

3.8.4 list_objects [-h hostname][-n host_num]

2. See the MPL Reference Manual for more information on language constructs.

"List_objects" lists active Mentat objects. Without arguments it lists all active Mentat objects (the list of hosts to query is determined by the current config file). The -h host_name option specifies which host to query, and the -n host_num option queries by number.

Usage:

```
elf% list_objects
Hostname[ 0]:          elf :          OBJECT CLASS          PID
                                      matrix_class          25480
Hostname[ 1]:          ash :          OBJECT CLASS          PID
                                      work_class            29186
Hostname[ 2]: beech :                 OBJECT CLASS          PID
Hostname[ 3]: larch :                 OBJECT CLASS          PID
                                      work_class            23958
Hostname[ 4]: gingko :                OBJECT CLASS          PID
Hostname[ 5]: oak :                   OBJECT CLASS          PID
elf% list_objects -n 1
Hostname[ 1]:          ash :          OBJECT CLASS          PID
                                      work_class            29186
```

3.8.5 kill_objects [-a][-h hostname [-a] [-p pid]][-n host_num [-a][-p pid]]

"Kill_objects" kills the specified objects. With the "-a" flag alone, all Mentat objects are killed. The "-h" and "-n" flags indicate a specific host. The "-a" flag can then be used to kill all Mentat objects on that host, or the "-p" flag can be used to kill only the specified object. The kill is a hard kill, peer objects communicating with killed objects will hang attempting to communicate.

Usage:

```
elf% kill_objects -h larch -p 23958
Hostname[ 3]:          larch
```

3.9.0 Miscellaneous Problems and Errors

1) Future stack overflow and underflow. If you fail to execute an "rtf()" for every Mentat member function invoked, (see code below), then you will eventually get the error

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

"FUTURE STACK OVERFLOW". On the other hand if you do more "rtf's" than the number of functions called, you will get the error "FUTURE STACK UNDERFLOW". This bug 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.

2) 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. They 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.

For a more comprehensive list of runtime errors, see Appendix “Errors”.

4.0 MentatView (A Mentat Run-Time System Monitor)

(MentatView was written by Gorrell P. Cheek, Dave A. Mack, and Virgilio E. Vivas)

4.1.0 Introduction to MentatView

It is often difficult to observe the execution of a parallel or distributed system because there is often no global knowledge maintained about each node's state. Furthermore, 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. It 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. MentatView shows the user which nodes are in the current Mentat configuration, how many Mentat objects are on each node and what the load and queue sizes are for each node in the system. This information can be used by the user for debugging, it can be used 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. A sample Mentatview session is shown in Figure 6. 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.

4.2.0 Using MentatView

The current version of MentatView runs using the Mentat run-time system on a network of Sun workstations running Unix and X Windows. To run MentatView, perform the following steps:

1) Have X Windows running and ensure that the Mentat RTS is running.

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 no display is specified, MentatView's window will appear on the machine where it is being executed.

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.

   • X performance Monitor - This option brings up an X Performance Monitor on any host the user specifies. A dialog box will appear prompting the user to enter the name of the host machine on which the performance monitor should run. This will not work if **xperfmon** is not supported in your environment.

   • (Un) Freeze - This command will stop the updating of the graphs in the MentatView window, allowing the user to freeze an instant of time. This is a tog-

gle command. Selecting it again unfreezes the MentatView window.

- 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.

Figure 6. Sample Mentatview display.

5.0 Library Classes

5.1.0 OOLIB

The object-oriented library (oolib) provides the Mentat user with a number of useful library classes that can be used by the application:

- `mentat_timer` : simple CPU stopwatch timer, useful for program timings
- `list_element` : ordered generic list class
- `cell_collection` : ordered collection of tagged list_element's
- `queue` : FIFO queue class
- `string` : variable-size strings
- `mem_handle` : general memory allocator
- `transportable_list` : dynamic variable-size memory-contiguous list structure

The oolib header file <oolib.h> can be found in /users/mentat/h. To use the oolib, you must include <oolib.h> in your Mentat class files and link with -lmentat.

5.2.0 Work_Manager

Work_manager is a dynamic job scheduler. It is initialized with the number of desired workers and the object name of the worker which is derived off of the worker class. Work_manager will enqueue vblock pointers to jobs to be done. It constrains the number of workers at any given time as well as making sure that the workers stay as busy as possible. It is possible to change the number of workers while feeding jobs in, if that is desired.

In order to have the work_manager perform jobs, the work to be done must be derived off of the worker class. The actual work to be done should be in a function called do_work(vblock*).

One example resides in the directory /users/mentat/examples/work_manager. It is a synthetic example called main. The worker is defined in /users/mentat/examples/work_manager/worker.c. The work_manager class is defined as follows:

```
persistent mentat class work_manager
{
        int max_num_workers;   //number of workers to keep busy
        int queue_size; // used for debugging purposes
        worker my_worker; // name of worker which will do the work
        job_list_element *my_queue; // the job queue
        int busy_workers; // number of workers actually busy (<= max_num_workers)
        int die_when_done; // boolean, TRUE if all jobs have been enqueued
        void register_work (int); // register that a job has finished and try to start more

public:
        void initialize(int num_workers, worker worker1);
        void enq_job(vblock *job_to_do);
        void num_workers(int new_num_workers);
        void wait_and_die();
};
```

The class work_manager provides the following functions:

void initialize(int num_workers, worker worker1)

This function initializes the number of desired workers, **max_num_workers** to **num_workers** and the object name, **my_worker** to **worker1**, which is derived off the worker class. The work manager will attempt to keep, at most , **max_num_workers** busy. Other member variables are also initialized at this time.

void enq_job(vblock *job_to_do)

This is the main part of the work_manager. Use **enq_job** to queue jobs in the work_manager which will send them to ready workers for execution.

void num_workers(int new_number_workers)

This function will change **max_num_workers**, the number of workers available to do work, to **new_number_workers**.

void wait_and_die()

This function lets the work_manager know that all jobs have been sent to enq_job by changing **die_when_done** to TRUE. When all enqueued and outstanding jobs have completed, the work manager will die. The worker class is defined as follows:

regular mentat class worker

{

public:

int do_work(vblock *work_to_do);

};

The class worker provides the following functions:

int do_work(vblock *work_to_do)

This function should be tailored to the specific application for which work_manager is being used to perform work. All information necessary to do the work should be contained in **work_to_do**.

Examples:

```
int main()
{
//A simple example of how work_manager can be used to manage jobs.
  work_manager working;
  vblock to_do;
  worker current_worker;
  working.create();
 // initialize work_manager to have 1 worker, and
 // to perform current_worker
  working.initialize(1, current_worker);

  //The next lines will enqueue 100 jobs for current_worker to do
  int i;
  for (i=0; i<100;i++)
    {
      working.enq_job(to_do);
    }
 //Wait a few seconds, then change the number of workers to 14
  sleep(10);
  working.num_workers(14);

   //Enquue another 100 jobs
  for (i=0; i<100;i++)
    {
      working.enq_job(to_do);
    }
 //Tell the work_manager that all jobs have been enqueued and it
 //can die as soon as it determines that all enqueued and out-
 //standing jobs have finished
  working.wait_and_die();
}
```

5.3.0 Mstreams

The Mentat I/O system is based on C++ streams and is available to applications that wish to enable this capability. It must be emphasized that I/O is not provided by the Mentat environment by default. Mentat classes that require I/O have to link in the Mentat stream library (libms) to enable stream I/O. Applications that do not need to do I/O do not incur the overhead of adding the Mentat I/O system to the environment. The Mentat stream interface hides its underlying implementation in terms of the Mentat file system (**mfilesys**) and Mentat file object (**mfileobj**) classes.

There are two basic problems that the Mentat I/O system takes care of: the Mentat stream library insures 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. The access rights problem arises since the Mentat run-time system creates class instances which will inherit the access rights of Mentat, and not of the user.

5.3.1 Enabling I/O

If a Mentat class wishes to do I/O, the header file <mstream.h> must be included in the source file associated with that Mentat class and linked with libms (-lms). There are four basic library routines that provide the basic stream I/O capabilities- MENTAT_OPEN_FS, MENTAT_CLOSE_FS, MENTAT_ENABLE_I/O, and MENTAT_DISABLE_I/O. For an application to correctly use the Mentat I/O system, several steps must be followed. First, the program entry point (main or mainloop) 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 done only 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_I/O (mfs) first. When I/O operations are finished, MENTAT_DISABLE_I/O (mfs) is called.

5.3.2 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 as described previously. If more general I/O is desired (i.e. file I/O), the Mentat I/O system provides an interface that supports user-defined file I/0. 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.

5.3.3 The Mfilebuf Class

To create user-defined streams, the user must first define the associated stream buffer object of type **mfilebuf** which takes the 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, ios::open_mode)

To open a file for reading use ios::in, for writing use ios::out, and for write-append use ios::app. Once a stream buffer has been bound to a specific file via **open** (the byte source/sink is now known), and 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 completed, the stream buffer needs to be explicitly closed:

mybuf.close ()

Appendix A contains a sample program that shows how the I/O system is used, and Appendix B contains a sample makefile for linking the I/O environment to Mentat class objects. 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 under Section “Unsupported C++ Features” and a work-around is described.

5.3.4 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 in the case that streams are misused. The most common problem is that the open on the mfilebuf failed for some reason. 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 try to insure 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 to force a synchronization where the close follows all I/O operations. If this doesn't work, the user may omit the MENTAT_CLOSE_FS , but must remember to kill the associated UNIX process (named mfs) when finished - kill_objects is recommended for this.

Appendix A-- Sample Program

```
#include <mstream.h> // include for mentat stream library
#include <appl.h>

main ()
{
 int res, i, x;
 float y;
 char c;
 APPL_CLASS appl_class_inst;

 // If the program desires to do IO, main must
 // declare the mfilesys object (mfs);
 // call MENTAT_OPEN_FS (mfs) and MENTAT_CLOSE_FS at the end.
 mfilesys mfs;
 MENTAT_OPEN_FS (mfs);

 // If a class (including main thread) desires to do IO
 // it must explicitly enable this capability
 MENTAT_ENABLE_IO (mfs); // gives us mcin, mcout, mcerr

 // Using standard streams is easy ...
 mcout << "Enter a number for x > " << flush;
 mcin >> x;
 mcout << "x = " << x << endl;
 mcin >> c;// Gobble up EOL
 mcout << "Enter a floating point number for y > " << flush;
 mcin >> y;
 mcout << "y = " << y << flush;
 mcin >> c;// Gobble up EOL

 // Files have be attached to streams in the following way ...
 mfilebuf in_buf (mfs);
 mfilebuf out_buf (mfs);


 // Create file streams
 in_buf.open ("in_file", ios::in); // mode is one of ios::in,out,app
 istream instr (&in_buf);

 out_buf.open ("out_file", ios::out);
 ostream outstr (&out_buf);

 // Using file streams ...
 for (i=1; i <= 100; i++) {
   instr >> c; // or in_buf.get(c)
   outstr << c; // or in_buf.put(c)
 }
```

Appendix B -- Sample Makefile

```
MPLC=$(MENTAT_BIN)/mplc
C++ = CC_saber
STREAM_DIR=/users/ecl2v/c++/work/tests/Saber_headers
SABER_DIR=/usr/include/Saber
CFLAGS = -I$(STREAM_DIR) -I$(MENTAT)/h -Dsun_mess \
   -DC++ -DBSD -DTIMES -DREG=register -O2
LFLAGS = -L$(MENTAT_LIB) -lmentat -lC -lm -lms -Bstatic

INCLUDE_DIR=$(MENTAT)/h
LIB_DIR=$(MENTAT_LIB)

ALL: teststr APPL_CLASS

teststr: sample_io.c APPL_CLASS.c
   $(MPLC) $(CFLAGS) sample_io.c
   $(C++) $(CFLAGS) sample_io.trans.c -o $@ $(LFLAGS)

APPL_CLASS: appl.h APPL_CLASS.c
   $(MPLC) $(CFLAGS) APPL_CLASS.c
   $(C++) $(CFLAGS) APPL_CLASS.trans.c -o $@ $(LFLAGS)
```