Mentat 2.6 Release Notes

The Mentat Research Group

Technical Report No. CS-94-07
February 17, 1994

Mentat 2.6 Release Notes

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.

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

February 17, 1994

Mentat 2.6 Release Notes

Explanation of changes made from Release 2.5

1.0 Changes for Release 2.6

- Performance Monitor

  MentatMeter, a performance monitoring tool, has been added to measure execution efficiency in Mentat applications.
- Guarded Statements

  Guarded statments are now supported using constants and local variables.
- Enhanced compiler support

  Mentat now supports GNU 2.4.5 C++.
- Library Enhancements - **MAY REQUIRE APPLICATION CHANGES**
- Support for Intel Paragon
- Bugs fixed
- FAQ's

2.0 MentatMeter

In the process of software development, software engineering dictates "Make it right and *then* make it fast". This applies to parallel programming as well. Since parallel programming is difficult, programmers spend a considerable amount of time ensuring that their program executes efficiently. In the last few years, one of the significant thrusts of research in software development has been to devise tools that make the development of efficient programs easier. These tools are called *performance monitoring and analysis tools*. The goal of performance monitoring and evaluation is to automate and simplify the task of analyzing the behavior of a program. While performance monitoring is the observation of the efficiency of a program, performance

evaluation attempts to measure the effects produced by a program designed in a particular manner. In addition to understanding the behavior of an application program, it is important to be aware of the benefits and limitations of the computing environment (machine and the run-time system) so as to optimize the program. The most effective way of providing this type of information to the user is through performance analysis and visualization tools that automatically collect performance information of a program during execution and display various characteristics of the program for the user to analyze. MentatMeter is a performance monitoring tool for Mentat, with which users can monitor the behavior of Mentat applications.

To aid performance monitoring, we have provided MentatMeter with the capability to measure the following metrics:

- wall-clock time and CPU time of member functions
- idle time of member functions
- number of invocations of member functions
- number of invocations of Mentat objects
- amount of data communicated between Mentat objects
- amount of message traffic generated by Mentat objects

With metrics such as wall-clock time, CPU time and function and object invocation counts, users can identify computational hot-spots in their programs. Such information tells users what percentage of the total execution time a particular member function has consumed. The user can then decide whether or not to optimize that member function. Metrics about the amount of message traffic and the communication pattern of an application indicate potential communication bottlenecks in a program. Users can then decide whether or not to optimize data structures for messages so as to reduce traffic.

In situations where no optimizations are possible, the information collected by MentatMeter helps users roughly estimate the costs of different parts of their program and then check to see if they match with actual costs. With the availability of such information, users will not spend too much time trying to optimize a poor algorithm when in fact no optimizations are possible. Instead, users can consider changing the structure of their program or sometimes even implement an entirely new algorithm.

The Mentat run-time system has been modified to collect performance data about an application during its execution. When the application finishes execution a file containing the performance data for the entire program is generated in the current working directory. This file can be quite large - on the order of a few megabytes for a medium sized problem. An example of a data file is shown in Figure 1.

Rather than expect the user to browse through the file manually, we have used a data visualization tool called Pablo, developed at the University of Illinois to view the performance data. The user must build a performance data flow graph using Pablo, for any or all of the performance metrics he or she wishes to view. This graph can then be executed. During execution, the graph takes the data file as input and plays back the values of the performance metric(s) during the course of execution of the entire application. Prior to data visualization, users can restrict the amount of data to be viewed by using a utility called `MM_filter`. If they desire to view the performance

Figure 1 **Performance Data File**

```
// Sample data file - gauss.ascii
#1:
// “description” “Procedure Entry Count Record”
“Procedure Entry Count”
        char “Classname”[];
        double “Timestamp”;
        int “Count”;
        int “Processor Number”;
        int “Process ID”;
        int “Function ID”;
};;
#2:
// “description” “Procedure Trace Record”
“Procedure Trace” {
        char “Classname”[];
        double “Timestamp”;
        double “Elapsed Time”;
        double “CPU Time”;
        int “Processor Number”;
        int “Process ID”;
        int “Function ID”;
};;
// more record descriptors
        .
        .
        .
// data records begin
“Procedure Entry Count” {
         [20] {
         “sblock”
}, 738345793671.420044, 1, 0, 1, 1 };;
“Procedure Trace” {
         [20] {
         “sblock”
}, 738345793671.420044, 4.570000, 0.000000, 0, 1, 1 };;
“Procedure Entry Count” {
         [20] {
         “sblock”
}, 738345793747.713989, 1, 0, 2, 1 };;
// more data records
        .
        .
        .
“Procedure Trace” {
         [20] {
         “sblock”
}, 738345812413.583008, 0.676000, 0.000000, 0, 1, 3 };;
// end of file
```

data of only a particular class or member function they can do so by specifying this in a “directives” file. `MM_filter` takes the directives file and the original data file and filters out the unnecessary data records, producing a new data file containing only the

required information. The original data file is left untouched for further use. A sample directives file is shown in Figure 2. Each line in the directives file consists of the type of data record to be viewed for a particular Mentat class and member function. The member function number is the order in which the public member functions of a Mentat class appear in the header file. The data file is in the Self-Describing Data Format (SDDF) as required by Pablo...

Figure 2 **Directives File**

```
“Procedure Entry Count” { “sblock”, 1 }
“Procedure Trace” { “sblock”, 3 }
“Object Entry Count” { “sblock” }
“Procedure Idle Time” { “sblock”, 1 }
“Message Length” { “sblock”, 3 }
“Proc Msg Gen Cnt” { “sblock”, 3 }
```

MentatMeter is currently available for applications running on a network of Sun workstations. In the future, it will be made available for other platforms as well.

To use MentatMeter, perform the following steps:

Step 1 Build the main program of the application with the **-meter** option. Note, when monitoring is not required, rebuild without the -meter option.

Step 2 Give the group “mentat” permission to write in your current working directory. Note, this is to allow MentatMeter to create the data file in the user’s directory.

Step 3 Ensure that the Mentat run-time system is running on a network of Sun workstations, and execute the application.

```
$ <executable_name> <command_line arguments>
```

When this step completes, a performance data file with the name <executable_name>.ascii is generated.

Step 4 [optional] Create a directives file containing requests for the type of performance information to be viewed.

Step 5 [optional] Execute MM_filter providing it with the name of the data file, the directives file and the output file.

```
$ MM_filter <executable_name>.ascii <directives_file>
<output_file>
```

Step 6 Build a performance data flow graph using Pablo.

Step 7 Execute the graph and view the performance metrics.

Note: When MentatMeter is executing, special persistent Mentat objects called _writer_instr are created to store the performance data to disc. These objects can be seen when a **list_objects** is executed or through MentatView. Currently, MentatMeter creates $\max\{2, \lfloor n/4 \rfloor\}$ _writer_instr objects, where n is the number of available

processors(nodes) in the current configuration. The user is advised to leave at least 3/4 of the available processors unused so that they can be dedicated to the _writer_instr objects. This will keep the perturbation caused by MentatMeter to the application down. When all processors have to be used by the application, the _writer_instr objects will be collocated with the application objects, and will cause more perturbation.

Steps 6 and 7 above have been elaborated in the next subsection.

2.1 Using Pablo for data visualization

Here we step through the process of using Pablo for visualization of performance data of Mentat applications. For further details about using Pablo, the reader is referred to the Pablo working documents [1], [2] and [3]. The reader is strongly encouraged to read [2] before using Pablo, as it contains detailed tutorials on how to use Pablo. These documents can be found in the Pablo distribution. In this example, we use Pablo to obtain the average execution time of all the member functions of a Mentat object in an application.

Step 1 Have X windows running on a Sun workstation.

Step 2 If Pablo has not been installed in your environment, do so by referring to the INSTALL files in the Pablo distribution, or contact mentat@virginia.edu.

Step 3 Set the appropriate environment variables to run Pablo.

Step 4 Next, enter the command:

```
$ runPablo &
```

A Pablo session is fired up and a window labeled Pablo will appear. First we must build a performance data flow graph and then execute it.

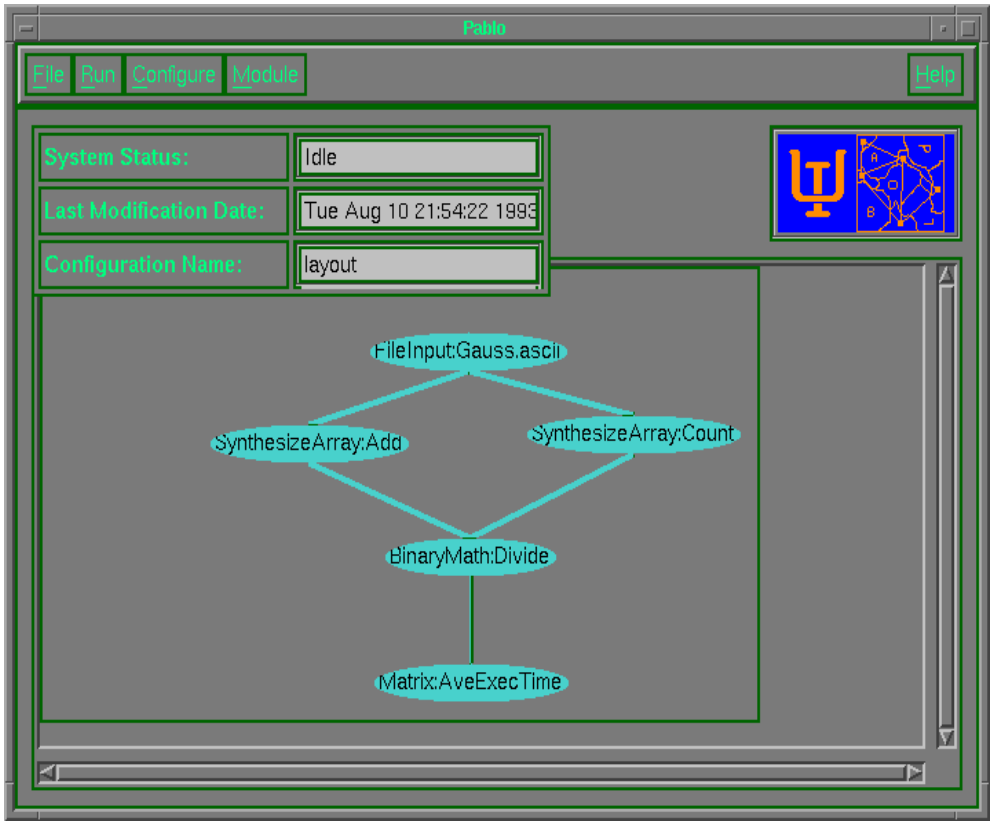
Step 5 Using the *Pablo Module Creation* window, add a *FileInput* module, two *SynthesizeArray* modules, a *BinaryMath* module and a *Matrix* display module.

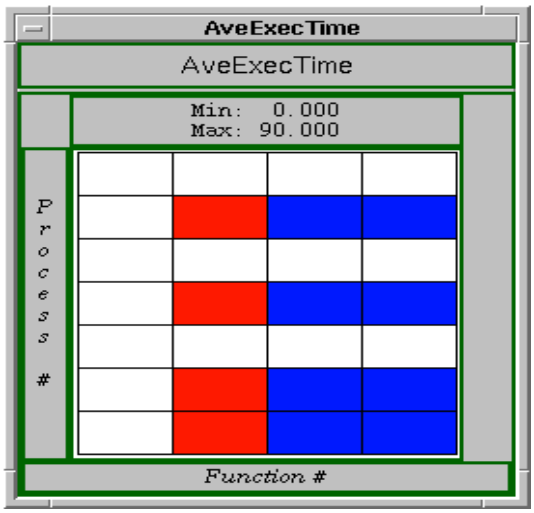
Step 6 Connect these modules as shown in Figure 3. Before execution, we must configure the graph, so as to tell each module what data it must process and output to the next module.

Step 7 Each of the modules can be configured as follows:

- the FileInput module to accept the performance data file
- the first SynthesizeArray module to extract fields from the “Procedure Trace” records and add the value contained in the wall-clock time field
- the second SynthesizeArray module to extract fields from the “Procedure Entry Count” records and keep track of the latest value of the “Count” field
- the BinaryMath module to divide the result from the first SynthesizeArray module with that of the second to obtain the average wall-clock time taken by each member function in the program

Figure 1 **Pablo Session for Execution Time of member functions of Mentat objects**





- and finally, the Matrix display module to display the array output by the BinaryMath module

Note: Several intermediate steps in the configuration process have been skipped in this document for brevity, and the user is once again referred to [2].

Step 8 Save the configuration.

Step 9 Run the graph. The Matrix display as shown in Figure 3 will be updated as the graph executes. The colors within the cells of the matrix will change as the values they contain change. These values can be viewed by clicking the mouse on the cells of the matrix.

2.2 References

Aydt, Ruth A., “The Pablo Self-Defining Data Format,” Pablo Working Documents, Dept. of Computer Science, UIUC, March 1992.

Aydt, Ruth A., “An Informal Guide to Using Pablo,” Pablo Working Documents, Dept. of Computer Science, UIUC, October 1992.

Reed, Daniel A., et al, “An Overview of the Pablo Performance Analysis Environment,” Pablo Working Documents, Dept. of Computer Science, UIUC, November 1992.

3.0 Guarded Statements

Some form of guarded statements are provided in many modern programming languages. Examples include the select/accept statements of ADA and guarded statements in CSP. Guarded statements permit the programmer to specify a set of entry points to a monitor-like construct. The guards are boolean expressions based on local variables and constants. A guard is assigned to each possible entry point. If the guard evaluates to true, its corresponding entry point is a candidate for execution. The rules vary for determining which of the candidates is chosen to execute. It is common to specify in the language that it is chosen at random. This can result in some entry points never being chosen.

The programmer may specify those member functions that are candidates for execution based upon a broad range of criteria. Further, the programmer may exercise scheduling control by using different priorities. The syntax for select/accept is shown below:

select_statement	::	**mselect** {guard_list};
guard_list	::	guard_statement; guard_list | guard_statement;
guard_statement	::	[guard]:[priority] guard_action;| :[priority] guard_action;
guard_action	::	**maccept** fct declarator; **break**; |
guard	::	Boolean expression based on variables, constants, and tokens.

Figure 3 **A Sample mselect/maccept Statement**

```
mselect {
      : maccept int func1(int arg1);
         break;
      (local_one > 5): maccept int func2();
         break;
}
```

Note: In the current implementation, guards must be local variables or constants. Further, priorities are not implemented.

The smelect statement has a similar semantics to the select statement of ADA. The availability of each guard-statement is controlled using a guard. The guards are evaluated in priority order. Within a given priority level each of the guards is evaluated in some non-deterministic order. Each guard is evaluated in turn until one of the guards is true; the corresponding member function for that guard is then executed. When the function has been executed, control passes to the next statement beyond the select.

A guard in Mentat is a boolean expression based on local variables or constants. Assignment statements are disallowed in guards (to prevent side effects).

4.0 Library Class Modifications - May require application changes

4.1 Operator new

The library classes **transportable_list**, **DD_array**, and **sparse_vector** have been modified to separate the operations of memory allocation and class object initialization. Thus, the operator **new** now requires arguments to indicate the amount of memory to allocate. For the classes derived from **DD_array**, the operator new must be called with the arguments specifying the number of rows and the number of columns. For the **sparse_vector** classes, the number of elements must be passed to the operator **new**. Figure 1 displays some examples of the use of the new operators. In addition, any user-

Figure 1 **Examples of operator new**

```
list = new(max_items, item_size) transportable_list(max_items, item_size);

real_array = new(rows, cols) DD_floatarray(rows,cols,data_ptr);

real_sparse_vector = new (num_elements) sp_sparsevector(dimension, num_elements)
```

defined classes derived from **DD_array** must specify their own version of the operator new.

4.2 User-defined variable-sized objects

All user-defined classes whose objects are of variable size must be derived from the base class **mentat_transportable_block**. This is to ensure that the objects are memory contiguous as required by the Mentat run-time system. Furthermore, each of these classes must overload the operator new. For example, assume that a class user_list has been derived off of **mentat_transportable_block**. In order to properly allocate enough memory for this object, the operator **new** must be overloaded in the derived class. The operator new must accept the total size of the object (in bytes) and the reserved parameter of **size_t**. In order to avoid compiler warnings, it is useful to derive a delete operator off of **mentat_transportable_block** as well. This operator need only take a void pointer as a parameter.

```
void *user_list::operator new(size_t s, int num_items) {

 return(mentat_transportable_block:: operator new (s,
 num_items*sizeof(item_type)));
}

void user_list::delete(void *ptr) {
 mentat_transportable_block::delete(ptr);
}
```

5.0 Bug Fixes

5.1 Communication System

Several improvements to the communication system have been made. First, in the event of an error, the message “socket mmps error” will be displayed once, and the application will likely hang.

In the event that an exception has occurred, the following error messages will appear: “floating point exception” or “segment violation.” As an aid in debugging, the name of the host on which the error appeared along with the name of class that caused the error will be displayed to the window in which Mentat was started.

5.2 Configuration Database Revisions

The configuration database has been improved with more error checking and more instructive error messages. In the event that a cluster is empty, the configuration manager detects and reports this error.

Further, the host naming facilities have been improved. Previously, the message passing system was unable to distinguish between two similarly named hosts, for example, the hosts maple and mapleleaf were considered to be the same host. Further, host names were case sensitive. These inconveniences have been removed. Hosts names can be of any case (even mixed-case).

5.3 Miscellaneous Compiler bugs

Several minor compiler bugs have been fixed. Many of the warnings have been eliminated, though many spurious "syntax error: undeclared identifier" messages remain. The compiler has also been extended to support the Mentat meter option discussed above.

6.0 FAQ's

There are several questions that we are asked again and again. This is usually caused by less than clear documentation. Sometimes though, the semantics are subtle. Most of the questions revolve around pointers. Below is the start of an ongoing FAQ file.

6.1 Passing pointers, their lifetimes and semantics

A user asks:

Please refer to page 16 of the Programming Reference Manual. Specifically, the section 10.0 Warnings. What does:

"reference and pointer arguments passed to Mentat class member functions are not preserved after the call. Consequently, the programmer must take care to first copy the arguments, if they are needed after the invocation"

mean? Is this description referring to the calling side? I am using the following piece of code:

```
string *tmp;
tmp = new string("10101010");
x.method(tmp);
delete tmp
```

Answer: None. For the caller there are no implications. There are implications for the callee. Suppose x is an instance of the persistent mentat class example_class, and that the class has the private member variable string *str; Further, suppose the body of method is:

```
void example_class::method(string *argument) {
 str = argument;
 rtf(0);
}
```

This code fragment would not work as expected because after the call to method the memory area pointed to by argument is returned to the heap manager by the Mentat run-time system. If the string is needed later, then it must be copied, e.g.,

```
void example_class::method(string *argument) {
```

```
 str = new string (argument);
 rtf(0);
}
```

6.2 Variable size arguments

We are often asked why code using variable size arguments does not work. The most common reason is that the variable used is an auto variable, not a heap variable. Auto variables have a constant amount of stack space allocated for them. Thus, they are not "variable size". To use a variable size variable, e.g., a variable derived off of mentat_transportable_block or DD_array, the variable must be a pointer, i.e., the data must reside on the heap. For example, DD_floatarray is a variable size class in the library. The code fragment:

```
DD_floatarray fred(5,5);
x.method(fred); // x is a mentat object
```

will not work because fred is an auto variable. In fact, the stack will be corrupted by using the constructor in that way. Instead, the following should be used:

```
DD_floatarray *fred = new (5,5) DD_floatarray(5,5);
x.method(fred); // x is a mentat object
```

The new(5,5) is the overloaded memory allocator being called.

6.3 ld.so call to undefined procedure <something> - on Suns

This is a known problem with the Sun OS dynamic linker. To avoid this problem we statically link our executables. Using the "file <filename>" command check whether your program is dynamically or statically linked. Mplc specifies -Bstatic for CC_saber and ATT, and -static for g++. If you add -Bstatic to the END of the compilation/link options (see the cc man page) the program will be statically linked, and the problem should go away.

6.4 Why don't my inline functions work?

Member functions for Mentat classes should not be implemented as in-lines. The compiler will sometimes generate correct code, but not always. The desired effect, fast function call, is NEVER realized, as the call is turned into a remote invocation by the compiler.

6.5 Can I have constructors for Mentat objects?

No. You can overload create though, which has a very similar effect. The one difficulty is that with the current release you cannot specify location hints and overload create for a Mentat class.

6.6 What are transportable lists? How can I use them?

Transportable lists are a handy base class that can be used to derive variable sized lists of user defined types. For example, suppose that I want to have a list of result structures, rstruct that I want to return from a Mentat object member function call.

```
struct rstruct
{
 int score0;
 int score1;
 int score2;
 unsigned sfnum;
 long lmark;
};
#define LIST_SORTED 1
class result_list:public transportable_list
{
 public:
 inline void *operator new(size_t sz);
 inline void *operator new(size_t sz,int max_count);
 result_list(int max_count) : (max_count,sizeof(rstruct)){}
 rstruct& operator[](unsigned i)
 {return *(rstruct*)(&transportable_list::operator[](i));}
 rstruct* append(rstruct* mo)
 {return (rstruct*)transportable_list::append((char*)mo);}
 void sort();
 int sorted() {return (get_flags() & LIST_SORTED);}
 int set_sorted() {set_flags(get_flags()| LIST_SORTED);}
};

inline void *result_list::operator new(size_t sz) {
 return transportable_list::operator new(sz);
}
inline void *result_list::operator new(size_t sz,int max_count) {
 return transportable_list::operator new(sz,max_count,sizeof(rstruct));
}



// use
result_list x = new (10) result_list(10;
x->append(value);
```

7.0 Known Bugs and problems

There are several known bugs that we have not had time to get to. There are work arounds for most all of these. If you know of any others, please let us know.

7.1 Will not work with ATT compiler version 3.X and later

This does not refer to the language version. The problem is that later versions of the compiler do not allow taking the address of a member function of a class and casting it to be a function pointer. The code generated by mplc uses that feature. The next (mentat

2.7) release will use a different technique to acquire the address of member functions. This does not effect g++ users though.

7.2 Unsigned short int, short int, etc.

The use of unsigned short int, short int, unsigned char, etc., as arguments to Mentat class member function calls is not supported. This is a compiler bug, and will be fixed in the next release. To get around this problem, use an int.

7.3 Non-intuitive behavior of calls on private members of Mentat objects

If a Mentat class member function invokes a member function on itself without using SELF, and that function performs an rtf(), and the caller performs an rtf(), a future stack underflow will occur. This is because the compiler does not generate code to create a dummy future for the call. **This will be fixed in the next release**. To avoid this problem partition your member functions into two classes, those that are called from the outside and which use an rtf(), and those that are only called locally, and do not use an rtf(). We admit that this is ugly, but it works.

7.4 Virtual Functions not supported on calls to Mentat objects

If an instance of a class that has virtual functions is passed as a parameter or return result to/from a Mentat object member function the virtual functions will not work unless the caller and the callee have the same executable. (It is safest to assume that they never work.) This follows from the address space disjoint nature of the computation model. We have designed a solution to this problem that will be incorporated into the next release.

7.5 No template or exception support

The MPL compiler predates the wide-spread availability of C++ compilers that support templates and exceptions. Thus, the language does not support these features. It requires a major re-write of the grammar to incorporate them. Therefore, do not expect template or exception support in the near future.

7.6 Default parameters for Mentat class member functions

Default parameters for Mentat class member functions do not work in this release. The compiler will generate a warning if no match can be found without the defaults, and the run-time system will generate an error,

“ERROR # args does not match: <pid> <operation id>:<argument count>”

when the number of arguments used is not what is expected.