Mentat Programming Language (MPL) Reference Manual

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

Computer Science Report No. TR-91-32
November 3, 1991

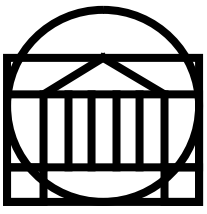
Abstract

Mentat Programming Language (MPL) Reference Manual

Andrew S. Grimshaw
Edmond C. Loyot, Jr.

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 Programming Language (MPL)
Reference Manual

Mentat Programming Language (MPL) Reference Manual

Release α.2

Andrew S. Grimshaw, Edmond C. Loyot, Jr., Jon B. Weissman
Department of Computer Science
University of Virginia
Charlottesville, VA
grimshaw@virginia.edu, ecl2v@virginia.edu

1.0 Introduction

One problem facing the designers of parallel and distributed systems is how to simplify the writing of programs for these systems. Proposals range from automatic program transformation systems that extract parallelism from sequential programs [1,2], to the use of side-effect free languages [3,4], to the use of languages and systems where the programmer must explicitly manage all aspects of communication, synchronization, and parallelism [5,6]. The problem with fully automatic schemes is that they are best suited for detecting small grain parallelism. The problem with schemes in which the programmer is completely responsible for managing the parallel environment is that complexity can overwhelm the programmer. Mentat [7-10] strikes a balance between fully automatic and fully explicit schemes.

There are two primary components of Mentat: the Mentat Programming Language (MPL) and the Mentat run-time system. MPL is an object-oriented programming language based on C++ [11] that masks the difficulty of the parallel environment from the programmer. The granule of computation is the Mentat class instance, which consists of contained objects (local and member variables), their procedures, and a thread of control. Programmers are responsible for identifying those object classes that are of sufficient computational complexity to allow efficient parallel execution. Instances of Mentat classes are used just like ordinary C++ classes, freeing the programmer to concentrate on the algorithm, not on managing the environment. The data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention. By splitting the responsibility between the compiler and the programmer we exploit the strengths of each, and avoid their weaknesses. Our underlying assumption is that the programmer can make better granularity and partitioning decisions, while the compiler can correctly manage synchronization. This simplifies the task of writing parallel programs, making the power of parallel and distributed systems more accessible.

This manual describes the MPL. We assume that the reader is familiar with the Mentat approach to parallel processing, and with the C++ programming language [11]. For more information on Mentat see [7-10]. To get an overview of the Mentat approach, complete with examples, see [9]. It is available via anonymous ftp from uvacs.cs.virginia.edu. The manual is designed to be

used in conjunction with the Mentat system distribution that includes an examples directory. The examples in the directory are complete running programs and can be used as templates when building your first Mentat applications. We recommend that you attempt some simple applications with Mentat before plunging into your application. This will give you experience using the language and the run-time system tools. In this document we will illustrate important points using code fragments as opposed to complete programs. The remainder of this manual is in seven sections. Sections 2 through 6 introduce the language and describe the language features. Section 7 discusses restrictions, and Section 8 briefly describes the examples.

2.0 The Mentat Programming Language

MPL is an extended C++ designed to simplify the task of writing parallel applications by providing parallelism encapsulation. Parallelism encapsulation takes two forms, intra-object encapsulation and inter-object encapsulation. In intra-object encapsulation of parallelism, callers of a Mentat object member function are unaware of whether the implementation of the member function is sequential or parallel, i.e., whether its program graph is a single node or a parallel graph. In inter-object encapsulation of parallelism, programmers of code fragments (e.g., a Mentat object member function) need not concern themselves with the parallel execution opportunities between the different Mentat object member functions they invoke.

The basic idea in the MPL is to allow the programmer to specify those C++ classes that are of sufficient computational complexity to warrant parallel execution. This is accomplished using the **mentat** keyword in the class definition. Instances of Mentat classes are called Mentat objects. The programmer uses instances of Mentat classes much as he would any other C++ class instance[1]. The compiler generates code to construct and execute data dependency graphs in which the nodes are Mentat object member function invocations, and the arcs are the data dependencies found in the program. Thus, we generate inter-object parallelism encapsulation in a manner largely transparent to the programmer. All of the communication and synchronization is managed by the compiler.

Of course any one of the actors in a generated program graph may itself be transparently implemented in a similar manner by a macro data flow subgraph. Thus we obtain intra-object parallelism encapsulation; the caller only sees the member function invocation.

MPL is built around four principle extensions to the C++ language. The extensions are Mentat classes, Mentat object instantiation, the return-to-future mechanism, and guarded select/accept statements. Each of these extensions is discussed in detail below.

3.0 Mentat Classes

In C++, objects are defined by their class. Each class has an interface section in which member variables and member functions are defined. Not all class objects should be Mentat objects. In particular, objects that do not have a sufficiently high communication ratio, i.e., the object operations are not computationally complex enough, should not be Mentat objects. To provide the programmer a way to control the degree of parallelism, Mentat allows both standard C++

1. The differences are described in Section 7, Restrictions. The primary difference is that parameter passing is by value.

classes and Mentat classes to be defined. By default, a standard C++ class definition defines a standard C++ object.

The programmer defines a Mentat class by using the keyword **mentat** in the class definition. He may further specify whether the class is **persistent** or **regular**. The syntax for Mentat class definitions is given below (Keywords are in **boldface**.):

new_class_def	::	mentat_definition class_definition |
		class_definition
mentat_definition	::	**persistent mentat** |
		regular mentat |
		mentat
class_definition	::	**class** class_name {class_interface};

Persistent and regular class definitions correspond to persistent and regular objects. Persistent objects maintain state information between member function invocations, regular objects do not. Thus, regular object member functions are pure functions. Figure 1 illustrates a Mentat class definition. A shared integer queue is a very fine grain object, and should only be used when

```
persistent mentat class shared_int_queue
{
    /* Note that private variables are truly private*/
    int head,tail,dsize;
    int *data;
 public:
    int deq();
    void enq(int);
    int count();
};
```

Figure 1. Mentat `shared_int_queue` class definition.

multiple Mentat objects need to share a queue. It is used here to illustrate Mentat class specification, and is not a good example of appropriate granularity.

Mentat classes are different from standard C++ classes in many ways. Each Mentat class is really two distinct entities: a front-end and a server-end. Instances of the front-end are called Mentat variables. Mentat variables are created when a user declares an instance of a Mentat class in his program. A Mentat variable is really a name that may point to an actual instance of a Mentat object. Instances of the server-end of a Mentat class are objects that actually implement the class. We call these instances Mentat objects. Mentat objects are independent objects: they are address space disjoint. Because they are address space disjoint, parameter passing is by-value. The ramifications are discussed in detail later.

Mentat variables are the mechanism through which programmers access and use Mentat objects. Mentat variables are C++ objects whose type is a Mentat front-end. Mentat variables contain a Mentat object name and incidental information and are typed according to the class of the Mentat object to which they point. Since Mentat variables are object names and not the actual rep-

resentation of their corresponding Mentat object, attempts to access private object data using the address of the Mentat variable will not succeed.

Mentat variables may be either *bound* to a specific instance of a Mentat object or *unbound*. Unbound Mentat variables do not address any particular instance. When unbound names are used for regular Mentat classes, the underlying system is free to choose an instance or create a new instance for each invocation of a member function. When an unbound name is used for a persistent Mentat class the system chooses an existing instance.

Each Mentat object (instance of a server-end) is disjoint from all other instances in the address space. The server-ends are implemented by the code of the Mentat class definition. Mentat objects may contain other non-Mentat objects and variables. The private data of the class definition is global and persistent to each instance. Each instance has an independent thread of control and a disjoint address space. It also has a system-wide unique name. The name is a Mentat object name and is available in the predefined Mentat variable SELF.

3.1 Mentat Object Instantiation & Destruction

3.1.1 Introduction

The issue of how to instantiate Mentat objects is an important one. Following the flavor and semantics of C++ would make it difficult to define generic instances. The difficulty stems in part from the fact that instances of Mentat classes are represented by their names and not by their physical selves. Only the name of the object is stored by the invoking object, not the instance itself. To illustrate this point, we consider an example in which there is a Mentat class *shared_int_queue*. There are three ways Mentat variables of *shared_int_queue* may be defined, static variables, auto variables, and heap variables. Furthermore, *shared_int_queue* could be a persistent or a regular class. We want to consider the problem of instantiating instances of *shared_int_queue* if we want to preserve the exact semantics of C++. In C++, static variables are instantiated at the start of the program, auto variables when the containing block of code is entered, and heap variables when they are newed or malloced. In Mentat these ways may not be appropriate for two reasons. First, regular objects may not ever need to be instantiated. For example, suppose that *shared_int_queue* is a regular Mentat class. Let *FRED* be defined to be of type *shared_int_queue*. If *FRED* is used often in expressions it may be better not to use the same instance each time. Instead we may want the underlying system to find or create instances at run-time, using a different instance for each invocation in order to increase parallelism. Therefore, we generally will want regular objects to be unbound to a particular instance. But if we instantiate a new instance as the usual rules call for in C++, using different instances would be incorrect.

Second, instead of creating new instances of a persistent class as called for by C++, it is sometimes more desirable to use the object as a name only, without instantiating a new version. For example, suppose there is a predefined file system object *file_system* that returns a *file_object* Mentat variable as the result of an open call. In the code fragment below we do not want the declaration of *A* to result in the instantiation of a new file object; we want to create an unbound Mentat variable.

```
A file_object; // Do not want a new file created.
```

```
A = file_system.open("user-data");
```

Our solution to this problem is to specify that when a new Mentat variable is declared, a new instance of the object class is not automatically instantiated. Instead, only an unbound object name of the appropriate type is instantiated.

3.1.2 Create()

The next issue to be considered is how the programmer creates new instances of Mentat objects. We have added four new reserved member functions for all Mentat class objects: `create`(), `bind`(). `bound`(), and `destroy`(). These functions are inherited from the base class Mentat and can be overloaded by the programmer of the class. The create function is used to instantiate new instances of Mentat classes. It takes as parameters user-provided initialization information. `Create`() also allows the user to specify where the new instance is to be instantiated, e.g., on a different processor, or on the same processor as some other Mentat object. The syntax is:

create_member	::	**create**(argument_list) |
		create(argument_list) (location_hints)
location_hints	::	**CO_LOCATE** obj_name |
		DISJOINT obj_name_list |
		HIGH_COMPUTATION_RATIO
obj_name_list	::	obj_name, obj_name_list |
		obj_name
obj_name	::	identifier

The programmer can specify location_hints, providing the underlying system with information that will be used in making instantiation decisions. The three values of location_hints are CO_LOCATE, DISJOINT, and HIGH_COMPUTATION_RATIO. CO_LOCATE tells the system to locate the object being created close enough to the object named by obj_name so that communication between the two is cheap. This usually means on the same processor. DISJOINT tells the system that the object to be created should be instantiated far away from any object in the obj_name_list because the object will usually be enabled in parallel with the objects named in the obj_name_list. It may not be possible to instantiate disjointly, if, for instance, every processor on the system has at least one object from obj_name_list on it. In that case the system does the best it can. HIGH_COMPUTATION_RATIO tells the system that the object to be instantiated has a particularly high computation ratio. The system can use this information to ensure that it is placed on a lightly loaded or powerful processor, even if the processor is very far away. The precise meaning of "close" and "far away" will vary depending on the algorithm used to make location decisions.

3.1.3 Bind() & bound()

Mentat variables may also be bound to an already existing instance using the inherited[2]

`bind(int scope)` member function, e.g.,

```
file_server fs; // Assume file_server is a persistent class
fs.bind(BIND_GLOBAL);
```

The integer parameter scope can take any one of three values, BIND_LOCAL, BIND_-CLUSTER, and BIND_GLOBAL. BIND_LOCAL tells the run-time system to restrict the search to the local host (the host may be a multiprocessor). BIND_CLUSTER tells the run-time system to search the entire cluster (subnet). BIND_GLOBAL tells the run-time system to search the entire system.

The `bound`() function indicates whether the Mentat object is bound to a particular instance. Bind and bound are often used together, e.g.,

```
file_server fs;
fs.bind(BIND_LOCAL);
if (!fs.bound()) fs.create();
```

Care must be taken when using this type of construct. Race conditions can develop.

3.1.4 Destroy()

The member function `destroy`() destroys the named persistent Mentat object. If the name is unbound, the call is ignored. Once destroyed, a Mentat variable cannot be reused. This is an implementation restriction that we expect to relax in the future. Care must be taken when using `destroy`(). If the name is in use by more than one Mentat object the "dangling pointers" problem occurs as when using pointers and the heap. An additional complication is that you may destroy the object before all operations applied to the object have completed.

4.0 The Return-to-Future Mechanism

The return-to-future function (`rtf`()) is the Mentat analog to the return of **C**. Its purpose is to allow Mentat member functions to return a value to the successor nodes in the macro data-flow graph in which the member function appears. There must be an `rtf`() for every member invocation. Failure to do so can cause deadlock, or, FUTURE_STACK_OVERFLOW.

Mentat member functions use the `rtf`() as the mechanism for returning values. The value returned is forwarded to all member functions that are data dependent on the result, and to the caller *if necessary*. A copy of the result is *not* sent back to the caller if it is not needed. In general, copies may be sent to several recipients.

While there are many similarities between `return` and `rtf`(), `rtf`() differs from a `return` in three significant ways. First, a `return` returns data to the caller. `Rtf`() may or may not return data to the caller depending on the data dependencies of the program. If the caller does not use the result locally, then the caller does not receive a copy. This saves on communication overhead. Second, a **C** `return` signifies the end of the computation in a function, while an `rtf`() does not. An `rtf`() indicates only that the result is available. Since each Mentat object has its own

2. All Mentat objects inherit a set of functions from the super class "mentat_object"; bind() and bound() are among these.

thread of control, additional computation may be performed after the `rtf`(), e.g., to update state information or to communicate with other objects. By making the result available as soon as possible we permit data dependent computations to proceed concurrently with the local computation that follows the `rtf`(). Third, in **C**, before a function can `return` a value, the value must be available. This is *not* the case with an `rtf`(). Recall that when a Mentat object member function is invoked, the caller does not block, rather we ensure that the results are forwarded wherever they are needed. Thus, a member function may `rtf`() a "value" that is the result of another Mentat object member function that has not yet been completed, or perhaps even begun execution. Indeed, the result may be computed by a parallel subgraph obtained by detecting inter-object parallelism.

For example, consider a transaction manager (TM) that receives requests for reads and writes, and checks to see if the operation is permitted. If it is permitted, the TM performs the operation via the data manager (DM) and returns the result. Below we illustrate how the read operation might be implemented. In a traditional RPC system, the record read would first be returned to

```
check_if_ok(transaction_id, READ, record_number);
// Assume that check_if_ok handles errors
rtf(DM.read(record));
```

the TM, and then to the user. In the above MPL code the result is returned directly to the user, bypassing the TM. Furthermore, the TM may immediately begin servicing the next request instead of waiting for the result and passing it back up. This can be viewed as a form of distributed tail recursion, or simple continuation passing.

5.0 Guarded Statements (Not implemented)

Some form of guarded statements are provided in many modern programming languages. Examples include the select/accept statements of ADA [12] and guarded statements in CSP [13]. 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, which results in less than optimal processor utilization.

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	::	**select** {guard_list};
guard_list	::	guard_statement; guard_list |
		guard_statement;
guard_statement	::	guard:guard_action;|
		guard:[priority] guard_action;|
		guard_action;|
		[priority] guard_action;
guard_action	::	statement-list; **break**;|
		accept fct declarator; statement-list;**break**; |
		test fct-declarator;statement-list;**break**;
guard	::	Boolean expression based on variables, constants, and tokens.

The select 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 the order of their priority. 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 statement-list for that guard is then executed. When the statement-list has been executed, control passes to the next statement beyond the select. Note that the fct-declarator portion of the guard_action determines the operation. The fct-declarator is not actually executed. It only provides information for the compiler.

There are three types of guard-actions: accepts, tests, and non-entries. Accept is similar to the accept of ADA. Non-entries are guarded statements. They do not correspond to a member function of the class. Tests are used to test whether a particular member function has any outstanding calls that satisfy the guard. When a test guard-action is selected, no parameters are consumed. Note that there is no "else" clause as in ADA. However, using the priority options, the user can simulate one by specifying that the clause is a non-entry statement and giving the guard-statement a lower priority than all other guard-statements. Then, if none of the other guards evaluates to true, it will be chosen.

Mentat guards are more powerful than guards in traditional languages. A guard in Mentat is a boolean expression based on local variables, constants, formal parameters of the member function being guarded, and message tag information such as the sender or computation tag. The

guard syntax is:

guard	::	(**NOMATCH** && guard1) |
		guard1
guard1	::	(guard1) |
		unary-operation guard1 |
		guard1 binary-operation guard1 |
		value
value	::	constant |
		variable |
		accept-variable |
		token-variable |
		expression
token-variable	::	arg-ident.arg-field
arg-ident	::	arg1 | arg2 |... | argN |
		accept-variable
arg-field	::	c-tag |
		from |
		priority |
		length

Guards are similar to expressions in C++, except that assignment statements are disallowed in guards (to prevent side effects), and accept-variables and token-variables are allowed in the expression.

Accept-variables are defined by the formal arguments of the member function of the accept or test which this expression guards. As such, an accept-variable is an identifier whose scope is the entire guard-statement in which it occurs. For example, in Figure 2, the accept-variable's account and amount are active in the entire code fragment. Token-variables are fields of the

```
(amount < 200) : accept withdrawal(int account, int amount);
    if account_exists(account)
          withdrawal(account, amount);
    else
          error(NO_SUCH_ACCOUNT, account);
    break;
```

Figure 2. Guarded accept statement.

arriving messages that are not part of the user data of the message, i.e., they are extra control information used by the underlying system. The token- variables are accessible in a read-only fashion by the applications programmer. There are four token-variables for each message, source, c-tag, priority, and length. The source field is the name of the Mentat object from which the token has come. The c-tag is the computation tag of the token. Priority is the priority of the token, and length is the length of the data part of the token. These fields are provided to the user so that finer

control may be obtained. However, it is not expected that they will be frequently used.

There is one set of token-variables for each accept-variable in the guard-statement. The token-variables have the same scope as accept-variables. They can be named by either their corresponding accept-variable name with a field name suffix, or by the argument number and a suffix, e.g., arg1.c-tag.

By default, tokens must have matching c-tag fields to be candidates for matching. This is accomplished by having an implicit (arg1.c-tag == arg2.c-tag ==... == argN.c-tag) ANDed to each guard. However, there are circumstances under which it might be desirable to circumvent this constraint. To do so the programmer adds the keyword NOMATCH as the first clause in the guard. Figure 3 illustrates the capability to match on fields other than c-tag. Each possible match-

```
struct srec1
{
     int account;
     .... // Some other stuff, application specific
};
struct srec2
{
     .... // Some stuff, application specific
     int account;
};
...
select
{
     (NOMATCH && (rec1.account == rec2.account))
           :accept debit(s_rec_1 rec1,s_rec_2 rec2)
     ...
};
```

Figure 3. Matching using other fields.

ing pair of tokens is examined to determine if it satisfies the guard. Note that it will likely involve much more overhead to check a guard when c-tag matching is turned off.

Priority is an integer ranging from -MAXINT to MAXINT. The default value is zero. There are two types of priority, that of the guard-statement, and that of the incoming tokens. The priority of the guard-statement determines the order of evaluation of the guards. It can be set either implicitly or explicitly. The token priority determines which call within a single guard-statement priority level will be accepted next. The token priority is the maximum of the priorities of the incoming tokens. Within a single token priority level, tokens are ordered by arrival time.

When a member function call is accepted, the current priority of the object is set to the priority of the tokens for the call. Any invoked subgraphs of the member function will have the same priority as the incoming tokens.

6.0 Parameter Passing

Mentat object member function parameter passing is call-by-value. All parameters are physically copied to the destination object. Similarly, return values are by-value. Pointers and references may be used as formal parameters and as results. However, the effect is that the memory object pointed to is copied. In the case of pointers the amount of data copied is determined by inspecting the class definition of the parameter (result). If the class has no `int size_of()` function defined, then `sizeof(class_name)` bytes are copied. If `size_of()` is defined, then it is invoked at run-time to determine the size of the actual parameter (result). The `size_of()` function **may not** be in-lined! While variable size objects are supported using the above mechanism, the object must be contiguous in memory[3]. The two examples below illustrate the specification and use of `size_of()`.

```
class string {
public:
 int size_of();
};
int string::size_of(){return(strlen(this)+1);}

class dblock {
 int num_bytes;
 char data[1];
public:
 int size_of();
 char &operator[](int loc){return &data[loc];}
 dblock (int size);
};

dblock::dblock(int size) {
 this = malloc(sizeof(int)+size);
 num_bytes=size + sizeof(int);
}
int dblock::size_of() {return num_bytes;}
```

Figure 4. Using variable size objects.

7.0 Restrictions

The address space independence between Mentat objects necessitates the imposition of five restrictions on Mentat classes. These restrictions derive from the fact that instances of Mentat

3. This restriction will be relaxed soon. The user will be permitted to specify a function, void marshall(char*); that will be used to marshall arguments.

classes are independent objects. All communication with and between Mentat objects is via parameters; there is no shared memory.

First, the use of static member variables for Mentat classes is not allowed. Since static members are global to all instances of a class, they would require some form of shared memory between the instances of the object. The preprocessor detects all uses of static variables and emits an error message.

Second, Mentat classes cannot have any member variables in their public definition. If data members were allowed in the public section, users of that object would need to be able to access that data as if it were local. Any use of such variables is detected by the preprocessor. If the programmer wants the effect of public member variables, appropriate member functions can be defined.

Third, programmers cannot assume that pointers to instances of Mentat classes point to the member data for the instance. The preprocessor will not catch this.

Fourth, Mentat classes cannot have any friend classes or functions. This restriction is necessary because of the independent address space of Mentat classes. If we permitted friend classes or functions of Mentat classes, then those friends would need to be able to directly access the private variables of instances of the Mentat class. Similarly, instances of a Mentat class cannot access each other's private data.

Fifth, it must be possible to determine the length of all actual parameters of Mentat member functions, either at compile time or at run-time. This restriction follows from the need to know how many bytes of the argument to send. Furthermore, each actual parameter of a Mentat member function must occupy a contiguous region of memory in order to facilitate the marshaling of arguments. Variable size classes must provide the member function `size_of`().

8.0 Warnings

There are a number of issues that MPL programmers must be aware of that can lead to unpredictable program behavior. First, 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 function invocation. Symmetrically, if a Mentat member function returns a pointer, the programmer must explicitly delete the reference when the function is finished using the value. The pointer is not deleted automatically when the function exits. If the programmer does not reclaim storage, memory leaks may result. Second, virtual functions do not work on Mentat class objects that have been explicitly passed as parameter arguments. The MPL programmer is advised to consult the Mentat User's Manual to determine the implementation status of several MPL features currently not supported.

9.0 Extended C++ Examples

The standard Mentat distribution comes with a set of example Mentat classes and applications. As of this writing there are three examples. You will find them in the directory "~mentat/examples".

The first example is a matrix solver that uses Gaussian elimination with partial pivoting. The application consists of two parts, a persistent Mentat class `sblock` that performs row reductions on a block sub-matrix, and a main program that uses `sblocks` to solve the matrix. The main program generates a test matrix, decomposes the matrix to the `sblocks`, and then iterates reducing the matrix and selecting the next pivot row. The example illustrates the specification of a persistent class, use of the class, and the use of a main program that uses Mentat objects. The directory includes a makefile that illustrates how to use the compiler. You may need to change the paths for the make to complete in your environment.

The second example uses a Mentat matrix class to perform matrix multiplication. It involves two Mentat classes, and a main program.

The third example is the traditional Fibonacci. The example is implemented with two regular Mentat classes, a fibonacci_class, and an adder_class. The adder class is used instead of a "+" operator because it illustrates tail recursion. The

```
rtf(adder.add(fib.fibonacci(n - 1), fib.fibonacci(n - 2)));
```

call allows the caller to exit and not wait for the result, reducing the number of objects that are instantiated at any given instant.

Perhaps the simplest example is time_rpc (see Figure 6) which we use to time the underlying communications system (the mentat_timer is defined in <oolib.h>, see Mentat User's Manual for a description). In this example, two instances of Mentat class generic, node1 and node2, are created with each instance calling a simple function one_arg during each iteration. A few things to note: for a given iteration, node1 and node2 will execute one_arg in parallel since they do not depend on each other; and, the assignments to k force a synchronization (the program must wait until the function calls complete before beginning the next iteration), see Figure 5. It is possible however, to pipeline across iterations.

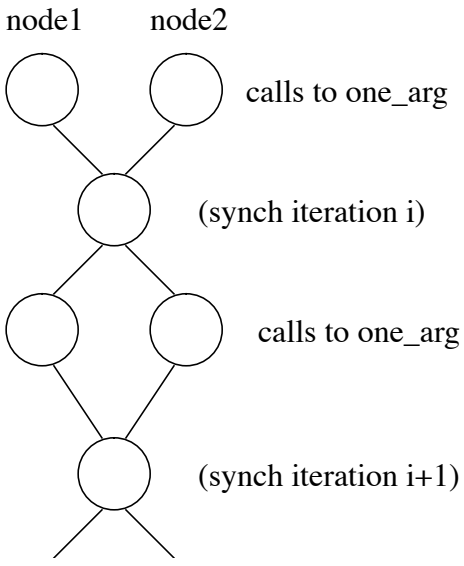


Figure 5. Dataflow graph for time_rpc

```
#include ...
mentat class time_rpc
{
public:
 int main_loop(string * arg);
};
int time_rpc::main_loop(string * arg)
{
 int iterations, delay;
 int i;

 sscanf (arg, "%d %d", &iterations, &delay);
 mentat_timer interval;
 generic node1, node2;
 node1.create(); // create the persistent objects ...
 node2.create();
 delay = 0;
 node1.set_delay(delay); // delay is used in member one_arg ()
 node2.set_delay(delay);
 interval.start(); // start the timer going ...
 for (i = 0; i < iterations; i++)
 {
     int j, k,l;
     j = node1.one_arg(delay);
     l = node2.one_arg(delay);
     k = j+1; // use evaluation of node1
     k = l+1; // use evaluation of node2
 }
 interval.stop(); // stop timer
 long elapsed = interval.msec();
 elapsed = elapsed / iterations;
 printf("Avg TIME = %d\n", elapsed);
 node1.destroy(); // destroy persistent objects ...
 node2.destroy();
 rtf(elapsed); // return value of elapsed
}
```

Figure 6. time_rpc example

10.0 References

[1] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "Cedar-A Large Scale Multiprocessor,"Proceedings of the 1983 International Conference Parallel Processing, pp. 524-529, IEEE, 1983.

[2] D. Kuck, R. Kuhn, B. Leasure, D. Padua, and M. Wolfe, "Dependence Graphs and Compiler Optimizations,"ACM Proceedings of the 8th Annual ACM Symposium on Princples of Programming Languages, pp. 207-218, January, 1981.

[3] W. B. Ackerman, "Data Flow Languages," IEEE Computer, vol. 15, no. 2, pp. 15-25, February, 1982.

[4] J. R. McGraw,"The VAL Language: Description and Analysis," ACM Transactions on Programming Languages and Systems, pp. 44-82, vol. 4, no. 1, January, 1982.

[5] G. R. Andrews, and F. B. Schneider, "Concepts and Notions for Concurrent Programming," ACM Computing Surveys, pp. 3- 44, vol. 15, no. 1, March, 1983.

[6] R. E. Filman, and D. P. Friedman, COORDINATED COMPUTING Tools and Techniques for Distributed Software, McGraw-Hill Book Company, New York, 1984

[7] A. S. Grimshaw, and J. W. S. Liu, "Mentat: An Object- Oriented Data-Flow System," Proceedings of the 1987 Object-Oriented Programming Systems, Languages and Applications Conference, ACM, pp. 35-47, October, 1987.

[8] A. S. Grimshaw,"The Mentat Run-Time System: Support for Medium Grain Parallel Computation," Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, SC., April 9-12, 1990, also available as, Department of Com- puter Science Technical Report TR 90-09, University of Virginia.

[9] A. S. Grimshaw, "An Introduction to Parallel Object-Oriented Programming with Mentat," University of Virginia, TR-91-97, April, 1991. Available via anonymous ftp from uvacs.cs.virginia.edu.

[10] A. S. Grimshaw, 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.

[11] B. Stroustrup, The C++ Programming Language, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[12] Reference Manual for the Ada Programming Language, United States Department of Defense, Ada Joint Program Office, July 1982.

[13] C.A.R. Hoare,"Communicating Sequential Processes," Communications of the ACM, pp. 666-677, August, 1978.