
Mentat Programming Language
Version 1.1
Reference Manual

The Legion Research Group

**Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
legion@virginia.edu
<http://legion.virginia.edu>**

Copyright © 1993-1998 by the Rector and Visitors of the University of Virginia.

All rights reserved.

Permission is granted to copy and distribute this manual so long as this copyright page accompanies any copies. The Legion system software herein described is intended for research and is available free-of-charge for that purpose. Permission is not granted for distributing the Legion system software outside of your site.

In no event shall the University of Virginia be liable to any party for direct, indirect, special, incidental, or consequential damages arising out of the use of the use of the Legion system software and its documentation.

The University of Virginia specifically disclaims any warranties, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. The software provided hereunder is on an “as is” basis, and the University of Virginia has no obligation to provide maintenance, support, updates, enhancements, or modifications.

Portions of the grammar used in the MPL front-end processor is Copyright © 1989, 1990 by James A. Roskind.

This work partially supported by DARPA(Navy) contract # N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), DOE D459000-16-3C and DARPA (GA) SC H607305A

The following people have contributed to the Legion project: Dawn Adelsberger-Mangan, Leo Cohen, Andrew Grimshaw, Adam J. Ferrari, Katherine Holcomb, Mark Hyett, Li-Jie Jin, John Karpovich, John Kingsley, Fritz Knabe, Marty Humphrey, Michael J. Lewis, Greg Lindahl, Ed Loyot, Mark M. Morgan, Anh Nguyen-Tuong, Charlie Viles, Christian Roberts, Thomas Spraggins, Sarah Wells.

1.0	Introduction	5
2.0	Background	7
3.0	The Mentat Programming Language	10
3.1	Mentat Classes	10
3.2	Mentat Objects	15
3.2.1	Using [Sequential] Stateful Mentat Objects	15
3.2.2	Two New Type Specifiers: <code>mentat_object</code> & <code>mentat_handle</code>	17
3.2.3	Using Stateless Mentat Objects	18
3.2.4	Other Pre-Defined Mentat Member Functions	19
3.2.5	Examples	19
3.3	Set Function Value Statement: <code>mentat_return</code> —Advanced Feature	22
3.4	IN, OUT, and INOUT Parameters	24
3.5	<code>mselect/maccept</code>	26
3.6	Parameter Passing	29
3.7	Warnings	36
4.0	Transitioning from Sequential to Parallel Programs	37
4.1	A Software Process Model	37
4.2	“Same Source” for Sequential and Parallel Implementations	38
4.3	Typical Legacy Code Transitions	40
5.0	Performance Optimizations	43
5.1	Loop-Splitting and Accumulators	43
5.2	Exploiting Tail Recursion	44
5.3	Stopping Parallel Recursion	45
5.4	Reducing the Number of Parameters	45
5.5	Using Stateless Object State	46
6.0	Summary	47
7.0	The MPL Compiler	48
8.0	Unsupported C++ Features	50
9.0	Programming Language Points	51
10.0	References	52

Mentat Programming Language

Version 1.1 Reference Manual

**A programmer's reference to
the Mentat object-oriented
parallel processing
environment**

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, to the use of side-effect-free languages, to the use of languages and systems where the programmer must explicitly manage all aspects of communication, synchronization, and parallelism. 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 strikes a balance between fully automatic and fully explicit schemes.

There are two primary components of Mentat: the Mentat Programming Language (MPL) and the Legion run-time system. MPL is an object-oriented programming language based on C++ [Stroustrup] 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. The manual is designed to be used in conjunction with the Legion 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 more complex applications. This will give you experience using the language and the runtime system tools. In this document we will illustrate important points using code fragments as opposed to complete programs. This manual is in nine sections. Sections 2, 3, 6 and 8, introduce the language and describe the language features, and Section 7 discusses restrictions. Sections 4 and 5 discuss porting, parallelization and performance issues. Section 9 is a bibliography.

2.0 Background

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. 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 nodes in a generated program graph may itself be transparently implemented in a similar manner by a 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 modifiers, the send ahead mechanism, and guarded select/accept statements.

The Mentat Philosophy on Parallel Computing

The Mentat philosophy on parallel computing is guided by two observations. First, that the programmer understands the problem domain of the application and can make better data and computation partitioning decisions than compilers can. The truth of this is evidenced by the fact that most successful production parallel applications have been hand-coded using low-level primitives. In these applications the programmer has decomposed and distributed both the data and the computation. Second, the management of tens to thousands of asynchronous tasks, where timing dependent errors are easy to make, is beyond the capacity of most programmers unless a tremendous amount of effort is expended. The truth of this is evidenced by the fact that writing parallel applications is almost universally acknowledged to be far more difficult than writing sequential applications. Compilers, on the other hand, are very good at ensuring that events happen in the right order, and can more readily and correctly manage communication and synchronization, particularly in highly asynchronous, non-SPMD, environments.

Intra-Object and Inter-Object Parallelism Encapsulation

A key feature of Mentat is the transparent encapsulation of parallelism within and between Mentat object member function invocations. Consider for example an instance `matrix_ops` of a `matrix_operator` Mentat class with the member function `mpy` that multiplies two matrices together and returns a matrix. As a user, when I invoke `mpy` in `X = matrix_op.mpy(B,C)`; it is irrelevant whether `mpy` is implemented sequentially or in parallel; all I care about is whether the correct answer is computed. We call the hiding of whether a member function implementation is sequential or parallel intra-object parallelism encapsulation.

Similarly we make the exploitation of parallelism opportunities *between* Mentat object member function invocations transparent to the programmer. We call this inter-object parallelism encapsulation. It is the responsibility of the compiler to ensure that data dependencies between invocations are satisfied, and that communication and synchronization are handled correctly.

Intra-object parallelism encapsulation and inter-object parallelism encapsulation can be combined. Indeed, inter-object parallelism encapsulation within a member function implementation is intra-object parallelism encapsulation as far as the caller of that member function is concerned. Thus, multiple levels of parallelism encapsulation are possible, each level hidden from the level above.

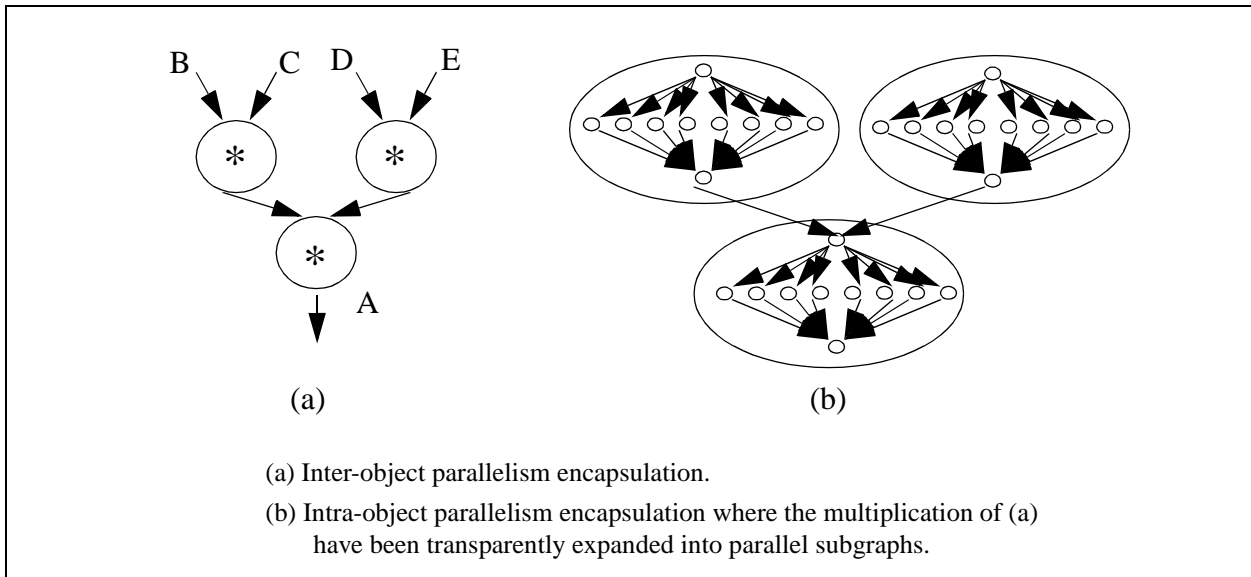


Figure 1

Parallel Execution of Matrix Multiply Operations

To illustrate parallelism encapsulation, suppose `X`, `A`, `B`, `C`, `D` and `E` are `matrix` pointers. Consider the sequence of statements

```
X = matrix_op.mpy(B,C);
A = matrix_op.mpy(X,matrix_op.mpy(D,E));
```

On a sequential machine the matrices **B** and **C** are multiplied first, with the result stored in **X**, followed by the multiplication of **D** and **E**. The final step is to multiply **X** by the result of **D*E**. If we assume that each multiplication takes one time unit, then three time units are required to complete the computation.

In Mentat, the compiler and run-time system detect that the first two multiplications, **B*C** and **D*E**, are not data dependent on one another and can be safely executed in parallel, as shown in Figure 4a. The two matrix multiplications will be executed in parallel, with the result automatically forwarded to the final multiplication. That result will be forwarded to the caller, and associated with **A**.

The difference between the programmer's sequential model, and the parallel execution of the two multiplies afforded by Mentat, is an example of inter-object parallelism encapsulation. In the absence of other parallelism or overhead, the speedup for this example is a modest 1.5:

$$\text{Speedup} = \frac{T_{\text{Sequential}}}{T_{\text{Parallel}}} = \frac{3}{2} = 1.5$$

However, that is not the end of the story. Additional, intra-object, parallelism may be realized within the matrix multiplication. Suppose the matrix multiplications are themselves executed in parallel (with the parallelism detected in a manner similar to the above). Further, suppose that each multiplication is executed in eight pieces (Figure 1b). Then, assuming zero overhead, the total execution time is $0.125 + 0.125 = 0.25$ time units, resulting in a speedup of $3/0.25 = 12$. As matrix multiplication is implemented using more pieces, even larger speedups result. The key point is that the programmer need not be concerned with data dependence detection, communication, synchronization, or scheduling: the compiler does it.

3.0 The Mentat Programming Language

3.1 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., whose object operations are not sufficiently computationally complex, should not be Mentat objects. Exactly what is complex enough is architecture-dependent. In general, several hundred executed instructions long is a minimum. At smaller grain sizes the communication and run-time overhead takes longer than the member function; resulting in a slow-down rather than a speed-up.

Mentat uses an object model that distinguishes between two “types” of objects, contained objects and independent objects.¹ Contained objects are objects contained in another object’s address space. Instances of C++ classes, integers, structures, and so on, are contained objects. Independent objects possess a distinct address space, a system-wide unique name, and a thread of control. Communication between independent objects is accomplished via member function invocation. Independent objects are analogous to UNIX processes. Mentat objects are independent objects. The programmer

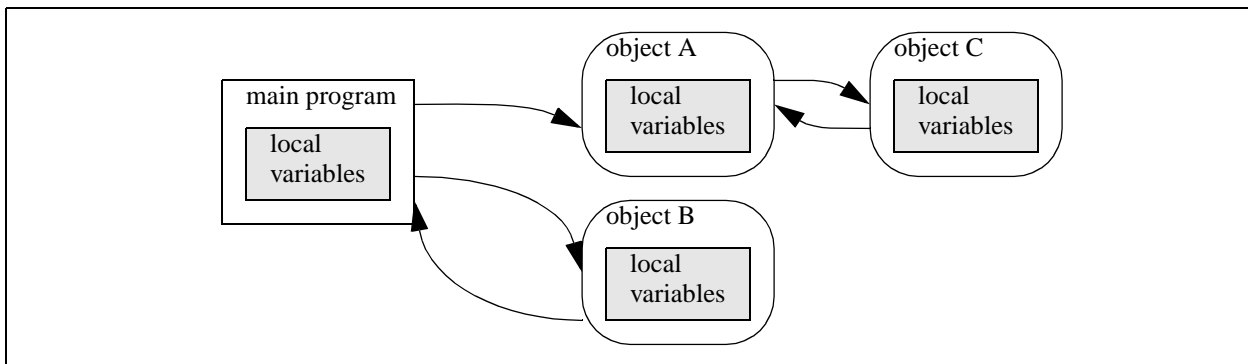


Figure 2 The Mentat object model. Mentat objects are address space disjoint and communicate via member functions and return values, shown here as directed arcs. Both the main program and Mentat objects may contain local variables.

defines a Mentat class by using the keyword **mentat** in the class definition. The programmer may further specify whether the class is **stateful**, **sequential**, or **stateless**. Instances of Mentat *classes* are called Mentat *objects*. The syntax for a class definition in Mentat is:

1. The distinction between independent and contained objects is not unusual, and is driven by efficiency considerations.

```
new_class_def::          mentat_definition class_definition |  
                        class_definition  
mentat_definition::    stateful mentat |  
                        sequential mentat |  
                        [stateless] mentat  
class_definition::     class class_name {class_interface};
```

Stateful and sequential objects maintain state information between member function invocations, while stateless objects do not.² Thus, stateless object member functions are pure functions. Stateless classes may have local variables much as procedures do and may maintain state information for the duration of a function invocation.



Mentat class interfaces may not have public member variables. The only way to modify or observe the state of a stateful or sequential mentat object is via member functions. It is illegal to directly access member variables. This restriction follows from the address-space-disjoint nature of mentat objects.

Example A

```
stateless mentat class integer_ops {  
public:  
    int add(int arg1, int arg2);  
    int mpy(int arg1, int arg2);  
    int sqrt(int val);  
};
```

The interface for the stateless mentat class **integer_ops** defines three functions that operate on integers. Note that they are all pure functions, the output depends only on the input. Because they are pure function and do not depend on any persistent state the system may schedule invocations of those functions on any processor in the system.

Example B

```
sequential mentat class integer_accumulator {  
    int running_total;  
public:  
    integer_accumulator(int initial_value);  
    void add(int value);  
    int current_value();  
};
```

The stateful mentat class **integer_accumulator** differs from the **integer_ops** class in that it does have state that persists from invocation to invocation. The state of the object is the member variable **running_total**. Once an instance of the class is created and the initial value assigned, subsequent calls to the **add** member function are applied to the same

2. Note that sequential objects are not supported by MPL, although they are used in examples throughout this manual.

object, incrementing the value of the `running_total`. The function `current_value` returns `running_total`.

Using Mentat Objects

Mentat objects are used much as C++ objects. The code fragment below uses the `integer_ops` and `integer_accumulator` classes to sum the squares of `n` integers.

Example C

```
{
  integer_accumulator A(0);           //Create an integer_accumulator
                                      //with an initial value of 0
  integer_ops B;

  for (int i=0; i<N; i++)
    A.add(B.mpy(i,i));
  cout << A.current_value();
}
```

In this example the loop is unrolled at run-time and up to `N` instances of the `integer_ops` class may execute in parallel. Note that parallel execution of the `B.mpy()` operation is achieved simply by using the member function. All of the `A.add()` operations are executed on the same object instance which is created and initialized with the `integer_accumulator A(0);` declaration.

The above examples illustrate the definition and use of Mentat classes. However, if we compiled and executed the above code fragments in Mentat, the parallel version of the program would execute far slower than an equivalent sequential program. The reason is that integer operations are of too fine a granularity for Mentat to exploit usefully. Larger grain operations are usually required for good performance.

Choosing Mentat Classes

This raises the question of when a class should be a Mentat class? A class should be a Mentat class if any one of four conditions is true:

- its member functions are computationally expensive,
- its member functions exhibit high latency (e.g., I/O),
- it performs a coordination role,
- it has shared state information (e.g., shared queues, databases).

Classes whose member functions have a high computation cost or high latency should be Mentat classes because we want to be able to overlap the computation with other computations and latencies, (i.e., execute them in parallel with other functions). Good examples are computationally expensive operations such as matrix multiplication of large matrices, finding objects in an image, or computing the configuration of a molecule.

I/O classes that support operations such as reading an image or searching a database are also good candidates for Mentat classes in that operations can be executed in parallel with other operations. For example we could read in a sequence of images from an image database and pipeline the subsequent computations.

Shared state objects should be Mentat classes because there is no shared memory in our model. Shared state between Mentat objects can only be realized using other Mentat objects. Rather than shared memory, Mentat supports a shared object space in which Mentat objects may invoke member functions on other Mentat objects to read and modify their state. Further, Mentat object member function parameters may be the names of Mentat objects.

Sometimes a class is a mentat class so that it can perform a coordination role between other Mentat objects. In our sum of squares example, the accumulator performs a coordination role of sorts. The summation could have been performed in the for loop itself, e.g.,

```
for (int i=0; i<N; i++)  
    j = j + B.mpy(i,i);
```

This implementation results in serial execution because the Mentat compiler generates code to block whenever the result of a Mentat operation (the **B.mpy(i,i)**) is used in a strict expression, i.e., the addition. Thus, the program will block on every iteration of the for loop, sequentializing execution. By using the accumulator Mentat object in a coordination role we eliminate the need to synchronize in the for loop. Another example of a coordination role is when a Mentat object contains other Mentat objects. For example, an image class may be composed of multiple sub-images. A member function to convolve the image would convolve each of the sub-images in turn. Thus, the image object acts in a coordination role.

Stateless, Stateful, and Sequential Mentat Classes

To illustrate the difference between stateless and stateful mentat classes, suppose we wish to perform matrix operations in parallel (e.g., a matrix-vector multiply). Recall that in a matrix-vector multiply a new vector is formed. Each element of the result is computed by performing a dot product on the appropriate row of the matrix with the vector) Because matrix-vector multiply is a pure function, we could choose to define a stateless mentat class **matrix_operators** as in Figure 3a. In this case, every time we invoke a **mpy()** a new mentat object is created to perform the multiplication and the arguments are transported to the new instance. Successive calls result in new objects being created with the arguments being transported to them.

Alternatively, we could choose to define a stateful mentat class **p_matrix** as in Figure 3b. To use a **p_matrix**, an instance must first be created and initialized with a **matrix**. Matrix-vector multiplication can now be accomplished by calling **mpy()**. When **mpy()** is used, the argument vector is transported to the already existing object. Successive calls result in the argument vectors being transported to the same object. In both the stateful and stateless case, the implementation of the class may hierarchically decompose the object into sub-objects and operations into parallel sub-operations. This is an example of intra-object parallelism encapsulation.

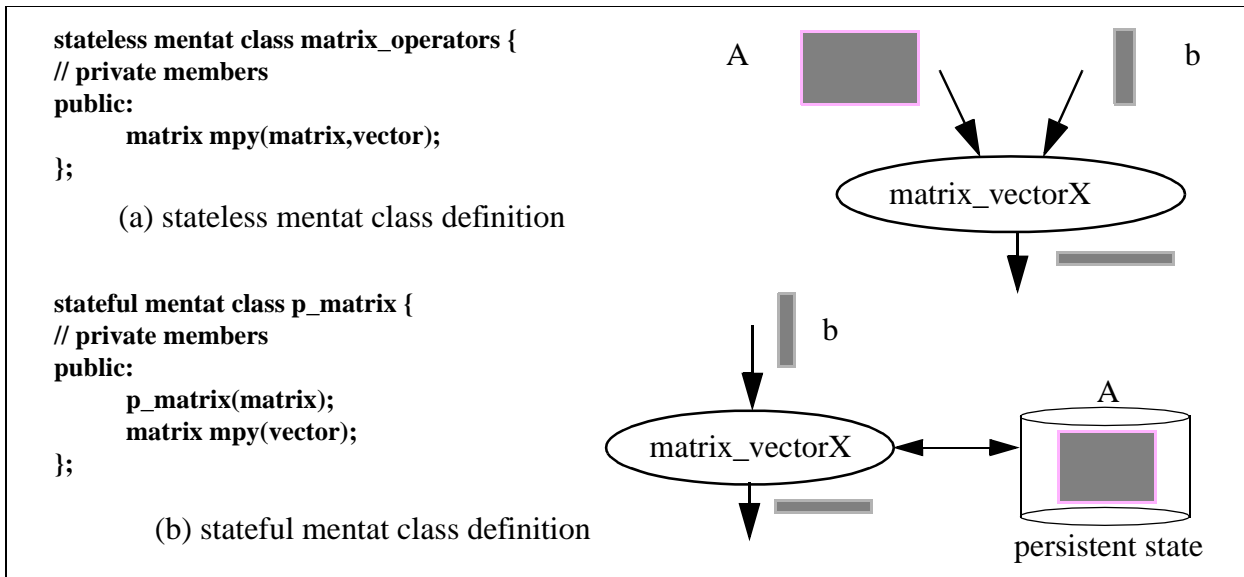


Figure 3

Matrix–vector multiply. (a) Using stateless objects. Note that both operands must be transported to the object. (b) Using a stateful object. Note that only the vector *b* is transported to the object. The matrix *A* is the state of the object and does not need to be transported.

The introduction of state significantly changes the program graph and resulting performance. The first difference between the two versions is the number of arguments. The stateful object version has only one argument, the vector, rather than two as in the stateless object version, the vector and matrix. The other “argument” is the state of the matrix object. Each distinct matrix will have its own state.

Stateful objects affect performance by reducing data communication. Suppose *A* is a 1024x1024 floating point matrix. *A* requires 4 MB of storage. To implement the matrix vector multiply as a stateless actor requires over 4 MB of communication, 4 MB to move *A* to the object, 4 KB to move *b* to the object, and 4 KB to transport the result. The stateful version requires only 8 KB, 4 KB for the argument and 4 KB for the result. This is a significant cost savings. In general the use of stateful objects can significantly reduce the amount of data communication at the cost of a possible loss of concurrency.

A sequential Mentat object is a special type of stateful Mentat object. A sequential Mentat object allows the user to control the order of invocation of member functions. Stateful Mentat object member functions are invoked as soon as all of their arguments are available, irrespective of their order within the caller. However, there are circumstances which may require a sequential ordering of the member function calls. For example, the order of push and pop operations on a queue can have a significant effect on the outcome of a program, particularly if the pop incorrectly arrives before the push! Sequential semantics are possible with a stateful Mentat object if the caller blocks after each member function invocation. This limits the amount of concurrency that can



be achieved by the application. A sequential stateful Mentat object will enforce sequential semantics automatically: the caller need not block after each invocation.

Sequential Mentat objects are not currently supported in this version of Mentat. They will, however, be supported in a future version of the compiler.

3.2 Mentat Objects

An instance of a Mentat class is a *Mentat object*. All Mentat objects have a separate address space, a thread of control, and a system-wide unique name. Instantiation of a Mentat object is the same as the standard C++ object instantiation semantics. Consider the code fragment:

```
{// A new scope
  int X;
  MC foo;                // MC is a stateful mentat class
}                        // end of scope
```

In C++, when the scope in which **X** is declared is entered, a new **integer** is created on the stack. In the MPL, because **MC** is a stateful Mentat class, **foo**'s data and code representation are automatically created (possibly on a different machine) and its name is bound to the locally created variable **foo**. When the scope is left, **foo** is destroyed along with the data and code that it was bound to. Though different in the sense that Mentat objects are distributed, their basic usage is the same as C++ objects.

There are important differences between Mentat objects and standard C++ objects however. Often, in Mentat, it is desirable to create objects that outlive their scope, such as a Mentat object that acts as a server object. In this example, an application may want to create this server object and leave it running so that other applications can use it. These other client applications must bind to the server object before they can use it. This is analogous to a C++ program wanting to use an object created in a different program; though C++ has no mechanism that can handle this, the MPL does.

Another difference between Mentat objects and standard C++ objects is that Mentat objects are distributed and hence cannot be referenced simply by providing a physical memory address (e.g. **&foo**). This difference is especially pertinent when passing Mentat object as function parameters. In C++ and the MPL, simply passing an object results in a copy of that object being made and that copy is used as the function's argument (i.e. pass-by-value). If the programmer wishes to avoid making a copy of the object, then C++ requires that the address of the object be passed instead (pass-by-reference is also allowed within C++ but this is simply syntactic sugar for passing pointers). Because Mentat objects cannot be referenced by physical memory addresses, simply passing the address of a Mentat object into a function is not sufficient. The MPL provides an extension to the C++ language that solves this problem; this solution is presented in section 3.2.2.

3.2.1 Using [Sequential] Stateful Mentat Objects

Mentat objects are used just like standard C++ objects in most cases. There are several operators that are defined for all stateful Mentat objects. These operators are used in the same fashion as their C++ counterparts but they **may not** be overloaded.

operator new()

This operator creates a new instance of the specified Mentat class. The created Mentat object may be placed (scheduled) on any host within the Mentat network.

```
operator new(MPL_enum_restriction how,  
             char *registeredName, MPL_enum_duration  
             duration=MPL_duration_permanent)
```

This operator creates a new instance of the specified Mentat class. The specified parameters restrict the placement (scheduling) of the new instance. The following enumerated types are defined:

```
enum MPL_enum_restriction {  
    MPL_restriction_none,  
    MPL_restriction_host,  
    MPL_restriction_vault  
}
```

```
enum MPL_enum_duration {  
    MPL_duration_permanent,  
    MPL_duration_oneTime  
}
```

MPL_enum_restriction specifies how the instance will be restricted (i.e., restricted to a particular host, vault, or not restricted). The host or vault to which the instance is restricted is identified in **registeredName**; the specified name is the name previously registered in the context space. **MPL_enum_duration** specifies whether the instance should be restricted to this placement forever or just during this particular activation.

Examples:

```
new(MPL_restriction_none, "" classname;  
    //same as new()  
new(MPL_restriction_host, "BootstrapHost") classname;  
    //create new instance located on  
    //BootstrapHost
```

MentatClass::MentatClass(MentatClass &)

(copy constructor)

Creates a new Mentat object and copies the data from the original Mentat object to the new Mentat object. The newly created object may be placed anywhere within the Mentat network.

MentatClass::MentatClass(...)

(*constructor*)

User defined constructors are allowed and may be overloaded. The copy constructor may **not** be overloaded.

~MentatClass::MentatClass()

(*destructor*)

The destructor destroys the Mentat object—removing it from the Mentat network. Currently, the destructor may not be overloaded by the user.

MentatClass::operator=(MentatClass &)

This operator copies the data from one Mentat object to another.

3.2.2 Two New Type Specifiers: `mentat_object` & `mentat_handle`

The MPL extends the C++ language by providing two additional type specifiers: **`mentat_object`** and **`mentat_handle`**. These type specifiers may be applied to [sequential] stateful Mentat classes only. The usual C++ syntax is used for these new specifiers.

Because of the distributed nature of Mentat, describing the location of a Mentat object within the MPL becomes more difficult than in C++. Simple pointers are no longer a sufficient method of providing information about locality. As with C++ objects, sometimes we wish to refer to the Mentat object itself and other times we wish to refer to the location (or address) of the Mentat object. To address this problem, the MPL uses the type specifiers **`mentat_object`** and **`mentat_handle`**. Mentat classes modified by the type specifier **`mentat_object`** have different operators defined than Mentat classes modified by the **`mentat_handle`** specifier. By default, if neither of these type specifiers are used to modify a Mentat class, then the type specifier **`mentat_object`** is assumed. All of the operators defined in 3.2.1 apply to stateful Mentat objects of type **`mentat_object MentatClass`** (which is the same as type “**`MentatClass`**”).

Mentat objects of type **`mentat_handle MentatClass`** have a different set of operators defined. Of most importance, objects declared as **`mentat_handle MentatClass`** do not automatically instantiate a new Mentat object nor do they automatically destroy an object when scope is left. They are analogous to C++ pointers for Mentat objects. The following are the only operators defined for objects of type **`mentat_handle MentatClass`**:

MentatClass::MentatClass(int scope)

(*constructor*)

This operator binds the object to an already existing instance of another object. This operator does not create a new instance of the class but instead initializes the object to refer to an existing object. The integer parameter `scope` can take one of three values: **`MPL_SEARCH_LOCAL`**, **`MPL_SEARCH_SUBNET`**, and **`MPL_SEARCH_GLOBAL`**. This restricts the search for an instance to the local host, the local subnet, or the entire Mentat network respectfully. Currently, only **`MPL_SEARCH_GLOBAL`** is allowed.

MentatClass::MentatClass(mentat_handle MentatClass &x)
(constructor)

This operator binds the object to an already existing instance of the class specified by the class instance parameter. This version of the constructor is identical to the following:

```
mentat_handle MentatClass foo =  
(mentat_handle MentatClass) MentatClassInstance
```

MentatClass::MentatClass(UvaL_Reference<Legion LOID> object)
(constructor)

This operator binds the object to an already existing instance of the class specified by the given LOID. This version of the constructor allows programmers to bind to instances created outside of the scope of the application. For example, an application could create an object, save the LOID to persistent storage, and terminate. That application (or a different application) could then read the stored LOID and bind to that object.

operator =(mentat_handle MentatClass &)

This operator copies the name binding from one Mentat object to another. Only the location of the object is copied not the data itself. This usage is similar to pointer assignment within C++: the address is copied not the data to which the address refers to.

operator mentat_object MentatClass()

This operator converts an object of type **mentat_handle MentatClass** to an object of type **mentat_object MentatClass**. Type conversion is safe and only affects the defined operators; the actual data does not change.

In addition to the operators defined in 3.2.1, the following operator is defined for objects of type **mentat_object MentatClass**:

operator mentat_handle MentatClass()

This operator converts an object of type **mentat_object MentatClass** to an object of type **mentat_handle MentatClass**. Type conversion is safe and only affects the defined operators; the actual data does not change.

Objects of type **mentat_handle MentatClass** are used predominately for creating objects that outlive their scope and for passing Mentat objects as parameters to functions. See 3.2.5 for examples of **mentat_object** and **mentat_handle** usage.

3.2.3 Using Stateless Mentat Objects

Mentat variables whose class is a stateless Mentat class are never explicitly created by the programmer via **new()** or constructors. Instead the programmer simply uses the

objects and the Mentat run-time systems logically creates a new instance for each member function invocation.

By default, Legion will instantiate one physical instance per stateless class. To add instances and achieve better performance users should use the command line tool **legion_stateless_add_workers**. To remove instances users should use the command line tool **legion_stateless_remove_workers**. When more than one actual instance exists for a given class the current scheduling policy randomly selects an instance. Future versions of Legion will give users more flexibility with respect to scheduling, e.g. round-robin or least-loaded.

```
legion_stateless_add_workers <class_name> <name1> <name2> ... <nameN>
```

For the stateless class <class_name> add workers named <name1>...<nameN>.

```
legion_stateless_remove_workers <class_name> <name1> <name2> ... <nameN>
```

For the stateless class <class_name> remove workers named <name1> ... <nameN>.

3.2.4 Other Pre-Defined Mentat Member Functions

```
int bound()
```

The **bound()** function indicates whether the mentat object is bound to a particular instance or not. This function is useful for checking whether or not a **mentat_handle** **MentatClass** object was successfully bound via the constructor.

```
UvaL_Reference<Legion LOID> loid()
```

The **loid()** function returns the Legion LOID of the instance to which the object is bound.

3.2.5 Examples

The following examples demonstrate Mentat object usage. The examples range from the very simple, typical usage to the very elaborate convoluted possibilities. These examples are meant to demonstrate what is possible, not necessarily what is desirable.

Example D

```
stateful mentat class MC {
  public:
    int mbrData
    MC();                               //overloaded default
                                        //constructor
    MC(int);                             //overloaded constructor
    int func();
    int func(int);
    void setValue(int);
```

```
    void getValue();
};

stateful mentat class Name_Server;

int typicalUsage() {
    MC foo;
    mentat_handle Name_Server bar(MPL_SEARCH_GLOBAL);
    int i;
    i = foo.func(bar.getInteger());
    printf("Result = %d\n", i);
} //NOTE: foo object has been
//destroyed, name_server object
//bar still exists.

int basicUsage() {
    MC a, b; //create and bind new objects
            //a & b
    mentat_object MC c; //create and bind new object c
            //(same as above)
    mentat_handle MC d, e; //create names only, no bind
    a = b; //copy a.mbrData to b.mbrData
    d = e; //copy name binding (from e to d)
    a = d; //copy name binding from d to a
    d = a; //copy name binding from a to d
    (mentat_object MC) d = e; //copy e.mbrData to d.mbrData
    (mentat_handle MC) b = a; //copy name binding from a to b
    a.func(); //invoke member function
    e.func(); //invoke member function,
            //run-time error if not bound
}

void moe(MC x) {x.setValue(99);}
void larry(mentat_handle MC x) {x.setValue(11);}
void curly(MC *x) {x.setValue(33);}
int callUsage() {
    MC a;
    larry(a);
    val = a.getValue(); //Value will be 11
    moe(a);
    val = a.getValue(); //Value will still be 11!
    curly(&a);
    val = a.getValue(); //Value will be 33
}

int newUsage() {
```

```
mentat_handle MC a(SEARCH_GLOBAL);
                                //create name and bind to
                                //existing object
mentat_object MC b(SEARCH_GLOBAL);
                                //error, constructor undefined

mentat_handle MC c, d;
mentat_handle MC *e;           //pointer to a mentat_handle MC
MC *f;                         //pointer to a mentat_object MC
MC g;

c = new mentat_handle MC; //error, incompatible types
e = new mentat_handle MC; //create new name and return
                                //pointer
f = new mentat_object MC;      //create & bind new object
                                //and return pointer
e = new mentat_object MC;      //create & bind new object.
                                //Will not auto-destroy on
                                //exit.
e = (mentat_handle MC *) (new mentat_object MC);
                                //explicit form of above
f = new (g) mentat_object MC;
                                //create new object on same
                                //host as g
f = new(MPL_restriction_host, "/hosts/foobar.cs");
                                //create new object on host
                                //foobar.cs
f = new(MPL_restriction_vault, "/vaults/some_vault");
                                //create new object on same
                                //host as vault some_vault
f = new(MPL_restriction_none); //same as new()
                                //not currently implemented...
                                //but defined
delete e;                       //delete front-end object only
delete f;                       //delete front & back-end
                                //objects
}

int scopeUsage() {
    MC a, b;
    mentat_handle MC c, z;
    {
        MC d, e;
        mentat_handle MC f, g, y;
        c = d;                   //copy the name
        c.func();               //call d.func()
        f = e;                   //copy the name
        e = (mentat_handle MC) a; //copy the name
        e.func();               //call a.func()
        g = (mentat_handle MC) b; //copy the name
    }
}
```

```
g.func();           //call b.func()
y = (mentat_handle MC) (new mentat_object MC)
                    //bind y to a new object
z = y;             // copy the name
}
d.func();          //error, d does not exist
b.func();          //Ok, call b.func()
a.func();          //error, a's back-end was
                  //destroyed by the
                  //implied 'delete e'
c.func();          //error, c is bound to d which
                  //was destroyed
z.func();          //OK, y's back-end was not
                  //destroyed
                  //NOTE: the original back-end
                  //object e has become an
                  //orphan!
}
```

3.3 Set Function Value Statement: `mentat_return`—Advanced Feature

The C++ language couples function value assignment with flow control in one statement: `return`. The C++ `return` statement terminates execution of the current function returning a value and control back to the caller. This is sufficient for sequential programs. However, in code designed for parallel execution, these semantics are not always desirable. Often it is advantageous to return the function value as soon as it is available even if more processing needs to be done within the function. Though the C++ language has no mechanism for this “send-ahead” semantic, MPL has extended the language to support “send-ahead” return values.

`mentat_return <expr>`

This statement immediately forwards the value of `<expr>` back to the successor nodes in the macro data-flow graph in which the member function appears. The returned value is forwarded to all member functions that are data dependent on the result and to the caller *if necessary*. In general, copies may be sent to several recipients. After the `mentat_return` statement has executed, the function continues normally until a `return` statement is reached.

The use of `mentat_return` is not required. If a `mentat_return` statement is not executed when the function terminates, then the value specified in the C++ `return` statement is forwarded to all successor nodes (and/or the caller). In general, the `mentat_return` statement should be used for send-ahead values and returning heap allocated variables that must be destroyed in the current scope. The next three examples illustrate `mentat_return`'s usage.

Example E

Consider a **stateful mentat class sblock** used in Gaussian elimination with partial pivoting. In this problem, illustrated in Figure 4, we are trying to solve for x in $Ax=b$. The

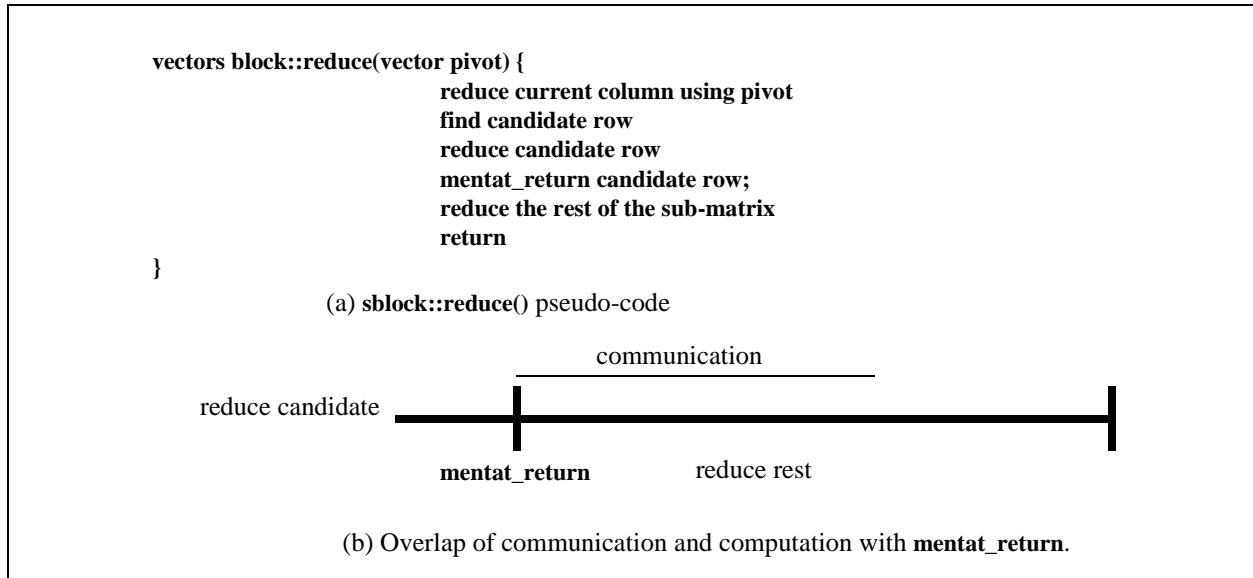


Figure 4

Gaussian Elimination with Partial Pivoting Illustrating the Use of `mentat_return`

sblocks contain portions of the total system to be solved. The **sblock** member function `vector sblock::reduce(vector);`

performs row reduction operations on a submatrix and returns a candidate row. Pseudo-code for the reduce operation is given in Figure 5a. The return value can be quickly computed and returned via **mentat_return**. The remaining updates to the sblock then can occur in parallel with the communication of the result (Figure 5b). In general, the best performance is realized when **mentat_return** is used as soon as possible.

Example F

Consider a transaction manager (**TM**) that receives requests for reads and writes and checks to see if the operation is permitted. When an operation is permitted, the **TM** performs the operation via the data manager (**DM**) and returns the result. Figure 5a illustrates how the read operation might be implemented. In an RPC system, the record read would first be returned to the **TM** and then to the user. In MPL the result is returned directly to the user; bypassing the **TM** (Figure 5b). Further, the **TM** may immediately

```

TM::read(int transaction_id, int record_number) {
    check_if_ok(transaction_id, READ, record_number);
    // Assume that check_if_ok handles errors
    mentat_return DM.read(record); //Note tail-recursive call
}

```

(a) Code fragment for Transaction Manager, `read()` member function



(b) Call graph illustrating communication for `TM::read()`

Figure 5

Tail Recursion in MPL

begin servicing the next request instead of waiting for the result. This can be viewed as a form of distributed tail recursion or simple continuation passing. In general, the “returned” graph may be arbitrarily complex—as in the matrix multiply example.

In this simple example, we wish to return the value of a heap allocated variable and then destroy that variable. Unlike standard C++, in the MPL this is possible.

Example G

```

myType MC::compute() {
    myType *returnValue = new(ASDF, 3) myType(42);
    mentat_return *returnValue; //forward myType to successors
    delete returnValue;       //clean up memory
    return myType;            //note: this value is never actually
                              //used since we have executed a
                              //mentat_return statement
}

```

3.4 IN, OUT, and INOUT Parameters

In addition to allowing the user to communicate values from mentat object member function calls via the return mechanism, MPLC also allows users to pass values back from mentat object function calls in the form of **OUT** and **INOUT** parameters (these are in addition to the **IN** parameter types, which are also supported and are the default parameter types). This is done in the same way that the compiler supports return values from mentat object function calls. That is, the compiler detects uses of the **OUT** or **INOUT** parameters after the function call and then attempts to block for the value to become available. The compiler also handles forwarding invocation requests to the

appropriate mentat objects, should the **OUT** or **INOUT** values be used as **IN** or **INOUT** parameters to other mentat object calls.

The **IN** parameter is the default parameter type for a parameter passed in to a mentat object function call. This parameter is packed into a LegionBuffer on the client side, and transported across the network to the server side of the call. There, it is unpacked and used as the appropriate parameter value in the mentat object function call. **INOUT** parameter types behave in exactly the same way, except that when either the member function in question exits or the user explicitly forwards the parameter value (see the description of the **forward_parameter** statement below) the value stored in the parameter at the time is packed into a LegionBuffer and returned to the caller (or forwarded to another object requiring its value). In this way, the user is able to pass values back from the mentat object through that function's parameters. Finally, **OUT** parameters behave exactly like **INOUT** parameters, except that their values are not packed into LegionBuffers on the sending side but rather are created (uninitialized) on the server side.³ This allows the programmer to send a value back through a parameter without paying the cost of packing and sending an unnecessary value on the initial function call.

As with the function value statement, **mentat_return**, there is also a mechanism for forwarding a value for an **OUT** or an **INOUT** parameter back to either the caller or an object to which the parameter invocation has been forwarded:

forward_parameter <parameter name>

This statement causes the value currently stored in the named parameter to be packed into a LegionBuffer and forwarded to waiting recipients. Like the **mentat_return** statement, this statement may forward multiple copies to waiting recipients, but after the values have been sent execution of the routine will continue as normal. In addition, should the programmer fail to issue a statement of this kind for any **OUT** or **INOUT** parameter, the parameter's value (at the time of method completion) will be automatically forwarded. Example H shows the use of this parameter mechanism to return values to callers.

Example H

A sample mesh calculation problem.

```
GridArray GridPiece::CalculatePiece(OUT int PieceNumber)
{
    PieceNumber = <calculate my number>;
    forward_parameter PieceNumber;           // We already have this value,
                                              // so forward it
    <do calculation of my grid>
```

3. Note that if a result (be it a return value or the result from an **OUT** or an **INOUT** parameter) is used as an **OUT** parameter in a call on a mentat object, the first result is lost. This involves cancellation of the original result invocation. If no other values from that first invocation are used in a blocking circumstance, then the original invocation is not guaranteed to occur.

```
    mentat_return AnswerGrideArray;           // We have the answer, so
                                              // return it

    <now do some clean up at the end>
}
```

(a) GridPiece::CalculatePiece() pseudo-code

```
PieceCollector Collector;
GridPiece Workers[NumWorkers];
GridArray Results[NumWorkers];

<do whatever initialization is necessary>

int i, WorkerNumber;
int BlockResult[NumPieces];
GridArray Answer;

for (i = 0; i < NumWorkers; i++)
{
    Results[i] = Workers[i].CalculatePiece(WorkerNumber);
    BlockResult[i] = Collector.CollectPartialResults(Results[i], WorkerNumber);
}

for (i = 0; i < NumWorkers; i++)
{
    if (BlockResult[i]);                       // Block on results
}

Answer = Collector.GetResults();
```

(b) Example Server-Side Code

3.5 mselect/maccept

The **mselect/maccept** construct of MPL is modeled on the ADA **select/accept**⁴. It is used to conditionally accept a subset of the member functions of the class. It can also be used to accept a member function call before the execution of the current member function is complete. When used in the context of an **mselect/maccept**, member functions are referred to as entry points or entries.

Each entry in an **mselect/maccept** may be controlled using a guard. The guards are boolean expressions based on local variables and constants. A guard may be assigned to each possible entry point. If the guard evaluates to true, its corresponding entry point is

4. We would like to have used **select/accept** as well. We could not because the Unix I/O libraries define a function **select**, which makes it impossible to use **select/accept** in MPL.

a candidate for execution. If more than one guard evaluates to true, the member function is chosen non-deterministically.

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 {<select_actions>}
<select_actions> ::       <guard_list> |
                          <guard_list> <else_action>
<guard_list> ::           <guard_statement> |
                          <guard_list> <guard_statement>
<guard_statement> ::     : <guard_action>; |
                          : [<priority>] <guard_action>; |
                          [<guard>] : [<priority>]
                          <guard_action>;
<guard_action> ::        maccept {fct declarator} break; |
                          maccept {fct-declarator}
                          <guard_stuff> break; |
                          <guard_stuff> break; |
                          break;
<guard_stuff> ::         {statement_list} |
                          {declaration_list} |
                          {declaration_list} {statement_list}
<priority> ::            {expression}
<else_action> ::         : else <guard_stuff> break; |
                          : else break;
```

<...> denotes non-terminals (defined here);

{...} denotes standard non-terminals (undefined here).

The `mselect` statement, an example of which is shown below,

Example 1

A sample `mselect/maccept` Statement

```
mselect {
    : maccept int func1(int arg1);
    break;
  [delay>0] : maccept int func2();
    break;
}
```

has 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-determin-

istic order. Each guard is evaluated in turn until one of the guards is true and there is a pending invocation on that function; 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.

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. The priority 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 priority levels 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 level as the incoming tokens.

A more complex example is the classic readers/writers problem. In Example J an implementation of the simplest of this problem is illustrated. Note that we have overloaded the default constructor of the class, and that the body of the constructor function never returns. Instead, it implements the server loop required for the readers/writers problem.

Example J

Readers/writers with `mselect/maccept`

```
stateful mentat class readers_writers {
    int num_readers, num_writers;
public:
    int init_readers_writers();
    int read();
    int write();
    int release_write();
    int release_read();
};

int readers_writers::init_readers_writers() {
    num_readers=num_writers=0;
    mentat_return 0;
    while (1) {
        mselect {
            [(num_readers==0)&&(num_writers==0)]:
                maccept write();
                break;
            [num_writers==0]: maccept write();
                break;
            :maccept release_read();
        }
    }
}
```

```
        break;
        :maccept release_write();
        break;
    }
}
return 0
}
int readers_writers::read() {
    num_readers++; return 1;
}
int readers_writers::write() {
    num_writers++; return 1;
}
int readers_writers::release_read() {
    num_readers--; return 1;
}
int readers_writers::release_write() {
    num_writers--; return 1;
}
}
```

3.6 Parameter Passing

Mentat object member function parameter passing is *call-by-value*. All parameters are physically copied to the destination object—pointer, array, and reference types may **not** be passed as a parameter to a Mentat member function. This is an artifact of MPL’s distributed nature, coupled with C’s inconsistent use of pointers to denote a single copy of a type as well as an array of elements of the base type. The distinction is up to the programmer to resolve, using context knowledge, convention, and documentation. In general, the compiler cannot distinguish the different uses. Return values from Mentat methods are also *call-by-value*.

Because of the distributed nature of Mentat objects, all passed parameters and return values must be “sent” between objects. Though MPL manages all of the communication, it still needs to “linearize” each parameter so that it may be sent through Legion’s communications layer. Linearization is the process wherein a data structure is placed in a contiguous block of memory. Delinearization is the reverse process. Consider Example K:

Example K

```
class string {
private:
    char *data;

public:
    string(char *x) {data = strdup(x);}
}
```

```
char *str() {return data;}
operator =(char *x) {data = strdup(x);}
operator =(string &x) {data = x.str();}
};
```

```
stateful mentat class Logger {
public:
    int printString(string x);
};
```

Without linearization, the Mentat method `printString()` would receive a copy of the `string` object whose `data` member (a pointer) would point to an address that was not valid within the scope of method (since the `Logger` class is an independent object running in a different address space). In order to address this problem, the MPL compiler will automatically insert code to call linearization and delinearization methods on all classes passed as parameters to Mentat member functions. These (de)linearization methods are member functions with the following signatures:

```
int pack(LegionBuffer &x);           //linearize
int unpack(LegionBuffer &x);        //delinearize
```

In many cases, the programmer is responsible for writing these methods. The MPL compiler will automatically call the `pack()` function on an instance when that instance is passed as a parameter to a Mentat member function. Code also will be inserted automatically to call the `unpack()` function before the code within the function is executed. Semantic meaning and correctness of the `pack/unpack` functions are the programmer's responsibility. However, the *intended* semantic meaning of `pack/unpack` is to (de)linearize the object. Example L demonstrates one possible definition of the `string` class with (de)linearization methods defined.

Example L

```
class string {
private:
    char *data;

public:
    string(char *x) {data = strdup(x);}
    char *str() {return data;}
    string &operator =(char *x) {data = strdup(x);}
    string &operator =(string *x) {data = strdup(x);}

    int pack(LegionBuffer &x) {
        int length = strlen(data) + 1;
        x.put(length);
        x.put_char(data, length);
    }
};
```

```
    }

    int unpack(LegionBuffer &x) {
        int length;
        x.get_int(&length, 1);
        data = new char[length];
        x.get_char(data, length);
    }
};
```

Example M shows another, very different, definition of **pack/unpack**

Example M

```
class string {
private:
    char *data;

public:
    string(char *x) {data = strdup(x);}
    char *str() {return data;}
    string &operator =(char *x) {data= strdup(x);}
    string &operator =(string *x) {data = x.str();}

    int pack(LegionBuffer &x) {
        x.put_char("Hello, World", 12);
    }

    int unpack(LegionBuffer &x) {
        printf("Hi, Unpack() being called.\n");
        compute(42);
    }
};
```

Both Examples K and L have perfectly valid definitions of **pack/unpack**, though example K has a semantic meaning more closely aligned with the MPL philosophy of **pack/unpack** usage.

The **pack/unpack** mechanism is a powerful and flexible feature. The complete program contained in Example N demonstrates how a binary tree can easily be (de)linearized.

Example N

```
class string {
  private:
    char *data;

  public:
    string(char *x) {data = strdup(x);}
    char *str() {return data;}
    string &operator =(char *x) {data = strdup(x);}
    string &operator =(string &x) {data = x.str();}

    int pack(LegionBuffer &x) {
      int length = strlen(data) + 1;
      x.put(length);
      x.put_char(data, length);
    }

    int unpack(legionBuffer &x) {
      int length;
      x.get_int(&length, 1);
      data = new char[length];
      x.get_char(data, length);
    }
};

class node;
class node {
  private:
    string *value;
    node *ltree, *rtree;

  public:
    node() {
      value = NULL;
      ltree = rtree = NULL;
    }

    node(char *s) {
      value = new string(s);
      ltree = rtree = NULL;
    }

    int insert(char *s) {
      if (strcmp(s, value-> str()) < 0)
```

```
        if (ltree)
            ltree->insert(s);
        else
            ltree = new node(s);
    else
        if (rtree)
            rtree->insert(s);
        else
            rtree = new node(s);
}
```

```
int pack(LegionBuffer &lb) {
    value-> pack(lb);
```

```
    if (ltree) {
        lb.put(1);
        ltree->pack(lb);
    } else
        lb.put(0);
```

```
    if (rtree) {
        lb.put(1);
        rtree->pack(lb);
    } else
        lb.put(0);
}
```

```
int unpack(LegionBuffer &lb) {
    int temp;
```

```
    value = new string;
    value->unpack(lb);
    lb.get_int(&temp,1);
    if (temp) {
        ltree = new node;
        ltree->unpack(lb);
    } else
```

```
        ltree = NULL;
    lb.get_int(&temp,1);
    if (temp) {
        rtree = new node;
        rtree->unpack(lb);
    } else
        rtree = NULL;
}
```

```
int print() {
    if (ltree)
```

```
        ltree->print();
        printf("%s\n", value->str());
        if (rtree)
            rtree->print();
    }
};

stateful mentat class foo {
    public:
        int gogo(node x);
};

int foo::gogo(node x) {
    x.print();
    return 1;
}

int main() {
    node *tree;
    foo A;

    tree = new node("house");
    tree->insert("bread");
    tree->insert("cat");
    tree->insert("monitor");
    tree->insert("zoo");
    tree->insert("rug");
    tree->insert("yoyo");
    tree->insert("lollypop");
    tree->insert("apple");
    tree->insert("nut");
    tree->insert("paper");

    tree->print();

    if (A.gogo(*tree));

    return 0;
}
```

The class **LegionBuffer** is a Legion Class that stores and retrieves information that can be sent to other Mentat (or Legion) objects. Amongst others, **LegionBuffers** support the following methods:

```
size_t put_char(char *data, size_t howmany);
size_t put_uchar(unsigned char *data, size_t howmany);
size_t put_short(short *data, size_t howmany);
size_t put_ushort(unsigned short *data, size_t howmany);
```

```
size_t put_int(int *data, size_t howmany);
size_t put_uint(unsigned int *data, size_t howmany);
size_t put_long(long *data, size_t howmany);
size_t put_ulong(unsigned long *data, size_t howmany);
size_t put_float(float *data, size_t howmany);
size_t put_double(double *data, size_t howmany);

size_t get_char(char *data, size_t howmany);
size_t get_uchar(unsigned char *data, size_t howmany);
size_t get_short(short *data, size_t howmany);
size_t get_ushort(unsigned short *data, size_t howmany);
size_t get_int(int *data, size_t howmany);
size_t get_uint(unsigned int *data, size_t howmany);
size_t get_long(long *data, size_t howmany);
size_t get_ulong(unsigned long *data, size_t howmany);
size_t get_float(float *data, size_t howmany);
size_t get_double(double *data, size_t howmany);
```

The semantics of these methods are to copy into (**put_xxx**) or out of (**get_xxx**) a **LegionBuffer** from/to the **data** **howmany** items of the specified basic type. **LegionBuffers** act like a streaming queue: the first data items placed in a **LegionBuffer** via **put_xxx** are the first data items retrieved via **get_xxx**. No type information about the inserted items is stored within the **LegionBuffer** so it is up to the applications programmers to correctly **put_xxx** and **get_xxx** the right items. The following examples will all work on a 32-bit architecture, but the results are very different. Assume that **A**, **B**, **C**, and **D** are **LegionBuffers**:

```
A.put_int((int *)data, 1);
A.get_int((int *)data, 1);
```

```
B.put_int((int *)data, 1);
B.get_float((float *)data, 1);
```

```
C.put_int((int *)data, 1);
C.get_char((char *)data, 1);
```

```
D.put_long((long *)data, 1);
D.get_char((char *)data, 1);
D.put_int((int *)data, 1);
D.get_char((char *)data, 1);
```

See the Legion Reference Manual for more information about **LegionBuffers**.

If a class used as a parameter to a Mentat method does not have a **pack/unpack** method defined, the MPL compiler may be able to generate automatically a **pack/unpack** function for the user. Whether or not the compiler can successfully (un)pack functions depends on several factors. If the MPL compiler cannot automatically generate these functions, then a fatal error message is produced. The MPL compiler takes a conservative approach to the automatic (un)pack function generations: 100% accurate (de)linearization is achieved or compilation is aborted. This conservative approach ensures program correctness with respect to (de)linearization.

The rules for (un)pack methods and automatic (un)pack generation are as follows:

- Only classes and structures passed as parameters to Mentat methods and Mentat classes themselves need to have (un)pack defined.
- Basic data types (int, char, float, etc.) do not need to be explicitly (de)linearized.
- Pointers, arrays, references, and unions may not be passed to Mentat methods—though they may be contained within classes.
- If a class passed as a Mentat method parameter contains both a **pack** and an **unpack** method and the function signatures of the (un)pack methods match the expected signatures, then MPL will use those methods for (de)linearization.
- If a class passed as a Mentat method parameter contains a **pack** or an **unpack** method (but not both) OR contains a **pack** and **unpack** method whose function signatures do not match the expected signatures, then MPL will generate a fatal compilation error.
- If a class passed as a Mentat method parameter does not contain a **pack** or an **unpack** method then MPL will attempt to automatically generate both a **pack** and **unpack** method for that class.
- Automatic generation of (un)pack methods will succeed if the following conditions are met:

The class does not contain any pointer, array, reference, or union member variables.

Any class (structure) member variables have (un)pack defined or can have the (un)pack methods automatically defined by these rules.

3.7 Warnings



Semantic equivalence to a sequential program is not guaranteed when stateful objects are used. This is trivially true for programs that have **mselect/maccept** statements; there are no equivalent serial programs. Mentat only guarantees that observable data dependencies are enforced. In order to ensure semantic equivalence, a sequential object should be used.

There are two cases to consider: (1) If the name of a Mentat object is passed to other Mentat objects and they access the object, then the order of access is not necessarily the same as in a sequential execution. (2) Repeated calls to the same stateful object are not necessarily executed in the same order if there is a data dependency between invocations. For example:

```
// Let A be a bound stateful Mentat object
int x, y;
x = A.op1(5);
y = A.op1(x);
```

will be executed in order. Assume **op2()** returns a void. The sequence:

```
A.op2(5);
A.op2(x);
```

may execute in either order. If the object **A** is defined as a sequential Mentat class, then the member functions will be executed in the expected order.

4.0 Transitioning from Sequential to Parallel Programs

There are two ways one can approach the process of transitioning from a sequential program to a parallel program using Mentat; you can take an existing application and add Mentat extensions, or you can start from scratch. In either case it is often desirable to have a program that, using compiler switches or `ifdefs`, can be executed either as a sequential program or as a parallel program. In this section we will discuss a software process model for implementing Mentat applications that we have found useful, how to write Mentat applications from scratch, some techniques for incorporating legacy code, and illustrate how one can write programs that can be either sequential or parallel.

4.1 A Software Process Model

The old adage that “there’s more than one way to skin a cat” certainly applies to the process of writing Mentat applications. There are as many ways to write programs as there are programmers. Our objective here is not to teach you how to write parallel programs. Rather it is to present a technique that we have found successful in reducing the time required and the difficulty. One final note: many of the techniques we describe are not unique to parallel programming or Mentat. They are generally applicable.

The first step is to decide which classes are candidates to be Mentat classes. Often at this stage it is very useful to first profile your application (if you have a legacy code) to verify your intuition about where the execution time is spent.

Often at this stage there are classes of objects, or functions, where the vast majority of computation is performed. It may be that the computationally heavy section is characterized by a large number of calls to a relatively simple function (i.e., the granularity of the individual calls is insufficient to justify parallel execution). This is often the case for example, when you have a large set of relatively simple objects, such as array elements. This type of problem is often called a data parallel problem.

If this is the case then additional classes may need to be developed that “contain” subsets of the smaller objects. For example, matrix operations are often computationally expensive yet the individual operations such as element-element multiplication are cheap. Rather than making the elements Mentat classes the matrix itself should be a Mentat class, with individual elements distributed to sub-matrices. For example, the matrix may be sub-divided by rows, columns, or by blocks, as shown in Figure 6.

The second step is to prototype the Mentat classes you will use and to build a skeleton implementation of the member functions that return dummy values. Compile and test this skeleton using your main program or a test-driver. The objective of this stage is to confirm that you are using the compiler and run-time environment correctly. Our experienced programmers usually skip the skeleton stage.

The third stage is to flesh out the bodies of the functions. In the case of legacy code this usually proceeds quickly. If the program is being written from scratch this stage is similar to what you would do if the program was a sequential program. We have found that testing each component in isolation as it is developed can significantly reduce application development time. We have found that it is often useful near the end of this stage to do performance debugging to verify the performance characteristics of Mentat classes.

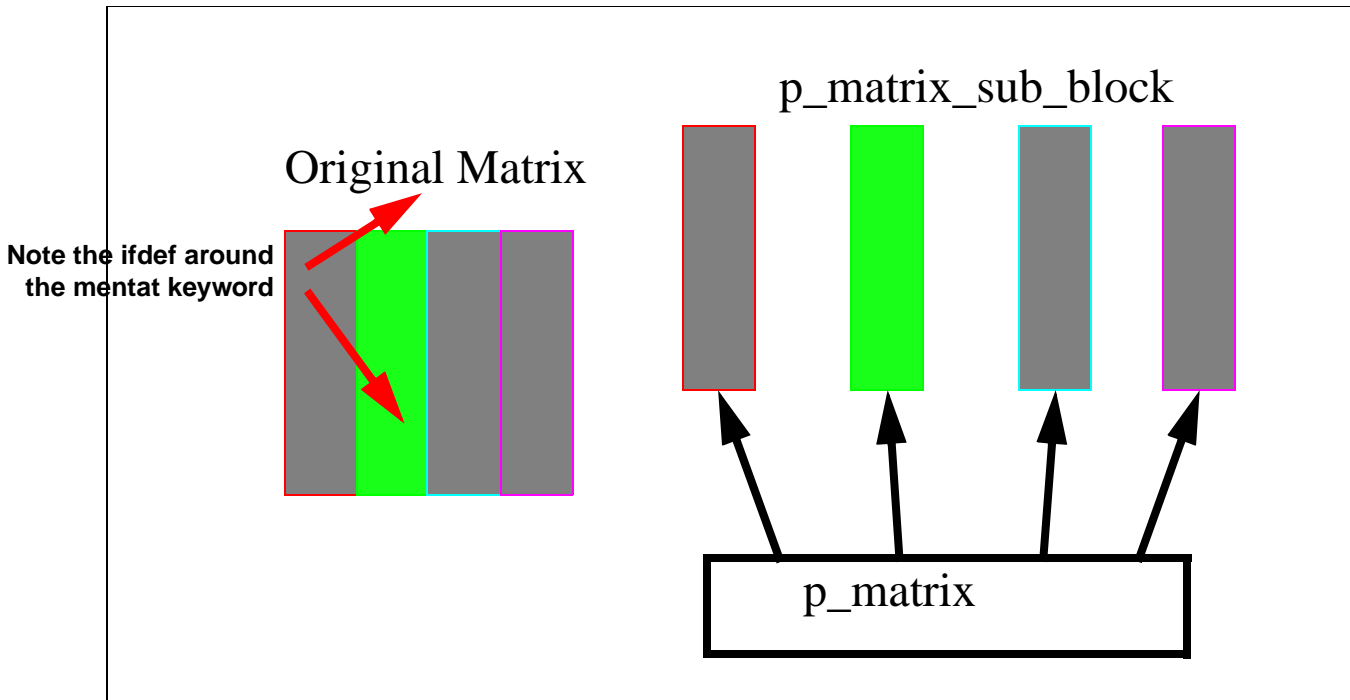


Figure 6 Block decomposition of a matrix. Each element is too small to be a Mentat class. Instead the matrix as a whole is a Mentat object. The matrix is implemented using multiple Mentat objects which are the sub-matrices. This is typical of a large class of data parallel objects in Mentat.

Finally there is whole application testing and debugging.

4.2 “Same Source” for Sequential and Parallel Implementations

One of the advantages of using Mentat is that for many applications it is possible to use the same source for both the parallel and sequential versions. This is the case for most of the applications that we have developed in house. The “same source” property is currently achieved using `#ifdefs` in the source code and different make targets in the make file. We anticipate that in a later release the compiler will directly support what must today be accomplished using `#ifdefs`.

To achieve a “same source” implementation the programmer must place `#ifdefs` around the MPL keywords `sequential`, `stateful`, `stateless`, `mentat`, and `mentat_return`. Further, the makefile must be updated for both cases. For example, Fibonacci can be implemented as shown below in Example O. Each Mentat class definition is `ifdef`’ed to be either a pure C++ class definition, or a Mentat class. Nothing else in the interface is different. Note that this is a compressed version of the actual code which can be found in the examples directory. All three `.c` files and the `.h` files are combined, and the main program has been compressed to its shortest possible form.

Example O

```
// *** adder_class.h
#ifndef SERIAL
mentat
#endif
class adder_class {
public:
    int add(int, int);
};
// *** fibonacci_class.h
#ifndef SERIAL
mentat
#endif
class fibonacci_class {
public:
    int fib(int);
};
// *** adder_class.c
int adder_class::add(int arg1,int arg2) {
    return arg1+arg2;
}
// *** fibonacci_class.c
int fibonacci_class::fib(int n) {
    adder_class a;
    fibonacci_class f;
    if (n <=1)
        return 1;
    else
        return a.add(f.fib(n-1),f.fib(n-2));
}
// ** main program file
int main (int argc,char ** argv) {
    fibonacci_class f;
    x = f.fib(10); // use of the fibonacci object
}
```

Note the ifndef



This technique for creating “same source” applications is widely applicable, not only to applications that use stateless Mentat objects as shown above, but also to applications that use stateful Mentat objects. If a stateful Mentat object is used, the programmer must keep in mind that the pre-defined operators are different than in C++. Thus, for example, the overloaded `new()` calls are no longer necessary and will need to be `ifdef`'ed out.

Example O can be re-coded as above.

“Same source” applications difficulties

```
{
    mentatClass *A, *B;

#ifdef SERIAL
    A = new(MPL_restriction_host, "/hosts/BootstrapHost")
    mentatClass;
#else
    A = new mentatClass;
#endif

    B = new mentatClass;
}
```

Not all applications lend themselves to easy “same source” implementations. Applications that use **mselect/maccept** are difficult to model using sequential semantics and hence are difficult to write using “same source” techniques. They are difficult to model using a sequential program because the order of execution of the object member functions depends on the internal state of the object. Yet during the sequential flow of the program, methods are invoked in the order determined by program flow. There is no easy way to simulate the **mselect/maccept** using a single thread of control and hence no easy transformation that we can recommend.

The second class of difficulties arises from the difference in parameter passing semantics—the semantics are always call-by-value in MPL, whereas they may be call-by-reference in C++. This is a particular problem with pointers. Because of the potential for memory leaks, heap corruption, or altered program semantics MPL does not allow pointers as either parameters or return values.

4.3 Typical Legacy Code Transitions

There is no typical Mentat application much as there is no typical application. In that sense the section heading is misleading. We have found though that there is one class of applications that is both very easy to parallelize and fairly common. These are problems of the form:

```
for all x in X do
    y = f(x)
    accumulate or do statistics on y
```

where the **for all** may be nested. The above pattern fits many different problems. For example, the program may be exploring a space of possible input parameters to a simulation in order to find the best parameter values. The function **f(x)** then is the simulation of a single point in the space.

A good example of this is a design tool used to explore the design space for solid state electron device developed in the electrical engineering department here at the University of Virginia. The main program is very simple. The main loop calls a member function **simulate** on a stateless mentat class **point**. The simulation function itself is a Fortran function that the researchers had previously developed. The details of the code, such as reading parameters and initializing variables, have been omitted for clarity. Only the basic structure remains.

```
// *** point.h
stateless mentat class point {
public:
    double Compute(FloatArray ins);
};

// *** point.c
extern void SIMULATE_(int *,double *,double *);

double point::Compute(FloatArray ins)
{
    // set up the variables for a Fortran call...
    // then call the Fortran function
    SIMULATE_(integer_ins,double_ins,Power);
    double result=Power[2];
    return result;
}

// *** the main program
main() {
    FloatArray ins;
    // initialization code...omitted
    x[0]=bases[0];
    for(i=0;i<points[0];i++){
        init->element(0,NCELL_OFF)=x[0];
        x[1]=bases[1];
        for(j=0;j<points[1];j++){
            init->element(0,NCELL_OFF+1)=x[1];
            x[2]=bases[2];
            for(k=0;k<points[2];k++){
                init->element(0,NCELL_OFF+2)=x[2];
                my_farray[index]=P.Compute(init);
                x[2] += strides[2];
            }
            x[1] += strides[1];
        }
        x[0] += strides[0];
    }
    // code to print results to output file...omitted
}
```

Another typical class of legacy codes includes those with persistent data structures, e.g., an array or mesh. These problems are more complicated to transition to a Mentat implementation, particularly if they are written in FORTRAN. Examples that include persistent state can be found in the examples directory of the Mentat distribution.

5.0 Performance Optimizations

It is not always sufficient to declare the appropriate classes as Mentat classes in order to achieve good performance. A correctly written application may run, generate the right answers, yet still not run quickly. It may not run quickly for many reasons: the granularity may be too small, there may be a sequential bottleneck, the program may be I/O bound, or the compiler may not be able to parallelize the code. Recall that the compiler and run-time system manage communication and synchronization for the programmer. However, they do so while attempting to preserve sequential semantics.⁵ The preservation of semantics may require that the compiler generate code to synchronize computation more often than is necessary. There are some constructs for which the compiler will generate sub-optimal code because it is being conservative. There are some simple transformations in these cases that can produce considerably faster code. This is a similar problem to that encountered using vectorizing compilers; some loops cannot be vectorized without first being modified. Below we look at four optimizations that may improve your application performance.

5.1 Loop-Splitting and Accumulators

The most common cause of poor performance is inter-iteration dependence in loops. This dependence can be broken in several ways. Below we will illustrate the problem and the solutions for a simple integer arithmetic class. The techniques discussed can be generalized to many forms of loops, including calls to arrays of stateful objects, and involving significant amounts of other computation (for example computing parameters) in the loop. Indeed, as long as the result is not needed in the loop to compute a value in the next iteration this technique will work. Consider the following loop, **B** is an instance of a stateless Mentat class **integer_ops** defined earlier.

```
integer_ops B;  
for (int i=0; i<N;i++)  
    j = j + B.mpy(i,i);
```

This loop will execute sequentially because we need the result of the **mpy** operation in order to carry out the addition. (The **+** is an integer **+**, not an overloaded Mentat member function.) There are several ways we can eliminate this dependence. First, we can split the loop by assigning the values into a temporary array. The result is that all of the **mpy**'s are executed in parallel.

```
integer_ops B;  
int *tmp = new integer[N];  
for (int i=0; i<N;i++)  
    tmp[i] = B.mpy(i,i);  
for (int i=0; i<N;i++)  
    j = j + tmp[i];
```

Another alternative is to accumulate the results using another stateless actor.

5. Sequential semantics may be violated when two different objects invoke methods on a third object. The order of invocation is not known.

```
integer_ops B;  
for (int i=0; i<N;i++)  
    j = B.add(j,B.mpy(i,i));
```

In this case the loop will completely unroll without blocking because we never use `j` in a strict expression. Instead it is only used as a parameter to other Mentat object member functions.

Finally, we can accumulate the results into a sequential Mentat object.

```
integer_accumulator A(0);           //Create an  
                                     //integer_accumulator with  
                                     //an initial value of 0  
                                     //integer_ops B;  
  
for (int i=0; i<N;i++)  
    A.add(B.mpy(i,i));  
j = A.current_value();
```

All of the above loop iterations will be run in parallel.

5.2 Exploiting Tail Recursion

Recall that Mentat objects are monitor-like and have a single thread of control. This means that if an object calls another object's member function and blocks waiting for the result, it cannot accept another member function invocation. There is of course the obvious possibility for deadlock if the called object invokes a function on the calling object. Performance may suffer as well if the caller could have safely been executing another member function in the interim. For example, suppose `A` and `B` are both stateful Mentat objects.

```
// a code fragment in the executing body of a member  
// function of object A  
x = B.op1();  
printf("x is %d\n",x);  
return(x);
```

Eliminate the printf to improve concurrency

The code fragment will block waiting for `x` so that it can be printed. If we eliminate the strict expression (the `printf`) on `x`, `B.op1()` will begin execution and return its value to where `A` was to return its value. At the same time `A` is now available to execute other member functions.

Similarly the `A` member function may require some computation or state update after the value `x` is resolved yet we still want to be able to accept other member function calls in the interim. This can be accomplished by "splitting" the member function. For example, `A` could have a member function `do_the_rest(int)` that completes the current member function and returns the same type.

```
x = B.op1();  
return(SELF*.do_the_rest(x));
```

The effect is that the result of the `B.op1()` call is passed to the `do_the_rest()` member function, which performs the rest of the function call and returns the value where

needed. The advantage though is that the object **A** may accept other member functions in the interim.

5.3 Stopping Parallel Recursion

In the Fibonacci example used in the standard distribution, recursion using Mentat objects continues until $n=1$. This results in a very large number of object instantiations even for modest numbers. Rather than recursively expand every call, we could stop the parallel recursion when n reaches some constant **MIN_RECURSE**.

```
int fibonacci_class::fib(int n) {
    adder_class adder;
    fibonacci_class f;
    if (n<2)
        return 1;
    else
        return adder.add(f.fib(n-1),f.fib(n-2));
                                // Note the tail recursion
}
```

To reduce the number of parallel calls we introduce the function **int lfib(int)** and call it when we have reached a small enough grain size.

```
int lfib(int n) {
    if (n < 2)
        return 1;
    else
        return lfib(n-1,n-2);
}
int fibonacci_class::fib(int n) {
    adder_class adder;
    fibonacci_class f;
    if (n < MIN_RECURSE)
        return lfib(n);
    else
        return adder.add(f.fib(n-1),f.fib(n-2));
}
```

Similar techniques can be used in many recursive decent algorithms, including search algorithms.

5.4 Reducing the Number of Parameters

There is overhead associated with each parameter to a member function. To reduce the overhead associated with member functions, multiple parameters may be clustered into a single parameter. A typical example is a member function that takes as parameters the boundaries of a rectangular region in a two dimensional space. The parameters **upper_left_x**, **upper_left_y**, **lower_right_x**, and **lower_right_y** could be passed as four separate parameters. Alternatively they could be passed as two points, each containing

an x and y coordinate, or as a single **region_class** that contains two points. This has the added benefit of often making the code more readable.

5.5 Using Stateless Object State

Stateless objects are, naturally, stateless. This means that the implementation cannot count on state persisting from one member function invocation to the next. However, all current implementations re-use stateless objects in order to improve performance. So while a new class instance is created on the stack for each invocation, causing each call to get its own member variables, the global variables are common to all of the calls which are executed on a particular processor. In other words, state can be saved in global variables, but it may not persist between invocations.

While you cannot count on the state being there, it can still be exploited to improve performance. Suppose that I have a stateless object member function that takes two parameters. The first parameter is the name of a large data file that contains problem-specific constant data. All member function calls for a particular program execution will have the same file name. The second parameter carries the non-constant data. For example,

```
for (i=0,N,i++)  
    tmp[i] = reg_obj.compute_image_at_time("my_world",i);
```

The first parameter contains the name of the file where the world database is stored. Each run of this program will use a different world database. The second parameter indicates a time-step for which to compute an image.

If we kept strictly to stateless object semantics then each invocation of **compute_image_at_time()** would need to re-read the world database. If the world database is large this could consume a large amount of resources and take extra time. If it is likely that successive calls will use the same world database this is a waste of resources.

To eliminate the waste, we can keep the name of the current world database and the current world database itself in global variables (**not** member variables). Then, on each invocation we can check the name of the world database parameter against the currently held database. If they are the same we do not need to re-read the database. If it is different, or we have never read in the database, then the database is read.

In practice we have found that this can lead to significant performance improvement.

6.0 Summary

Class definitions in Mentat:

```
new_class_def::      mentat_definition class_definition |
                    class_definition
mentat_definition::  stateful mentat |
                    sequential mentat |
                    stateless mentat
class_definition::   class class_name {class_interface};
```

Mentat member function invocation:

```
<Mentat_class_variable>.<member_fn> (args)
```

The semantics of Mentat member function invocation is *call-by-value* parameter passing.

new(...):

The `new(...)` operator creates a stateful Mentat object and returns the binding.

```
new() <Mentat_Class>;
```

```
operator new(MPL_enum_restriction how,
             char *registeredName,
             MPL_enum_duration
             duration = MPL_duration_permanent)
```

The Mentat class `<Mentat_Class>` must be **stateful** or **sequential**.

Bound():

The `bound()` member function call determines if `<Mentat_cv>` is bound to an instance: returns 0 or 1.

```
<Mentat_cv>.bound()
```

Loid():

The `loid()` function returns the Legion LOID of the instance to which the object is bound.

```
UvaL_Reference<LegionLOID> loid()
```

SELF:

The keyword **SELF** is a system-defined name that the “current” Mentat object is bound to. It is available within Mentat member functions; the Mentat analogue of “this” in C++.

7.0 The MPL Compiler

Object Paths and Current Context

In the current version of MPL, the MPL compiler locates its Mentat class objects in Legion context space in order to take advantage of Legion's object persistence. Therefore, all Mentat class objects used in MPL programs must have a Legion context name (please see Legion documentation for information on context space). When the program wishes to create an instance of a Mentat class, it uses a search path, which contains a set of possible context paths that may lead to the class object. This search path is created at run time.

There are two opportunities to specify search paths, one at compile time and one at run time. The first opportunity, at compile time, lets users name those context paths which might lead to the class object. To do this, use the **-LegionL** compiler option flag with the **legion_mplc** command in order to specify one or more possible context paths. This flag is similar to the **-L** and **-I** flags used by many C and C++ compilers although its syntax is slightly different (see below for more information). At run time, the specified paths are searched in the order that they were listed on the command line. The search path resulting from this method is used only by the object being compiled.

The second opportunity is to specify a path at run time. This method is not valid for stateless objects, as they are shared by multiple users and have start times that are prior to applications' start times. To specify a run time path, the user puts possible context paths in the `LEGION_OBJECT_PATH` environment variable: when an MPL program is started the program retrieves a second search path (the first being the path created at compile time, above) from the user's current environment. The `LEGION_OBJECT_PATH` environment variable and the user's normal Unix search path variable (`PATH`) use identical syntax, but the former specifies a path in context space. At run time, the program automatically checks the `LEGION_OBJECT_PATH` environment variable for a second search path. If it finds a second search path it passes it to all of its stateful objects and its stateful objects' stateful children instances, so that they share a common search path.

When a Mentat object wishes to instantiate another Mentat object, the parent object searches the compile time search path first. If this search path fails (i.e., does not lead to the desired object), the parent object tries the second search path, specified in the user's environment variable (`LEGION_OBJECT_PATH`). If neither of these search paths work, the parent object will look for a matching class in the user's current context, even if the current context was included in the two search paths.

The context path names used in the search paths can be relative or absolute.

Using legion_mplc

The usage of the **legion_mplc** command line tool is:

legion_mplc [*<options>*]

where *<options>* are any standard C++ compiler options (such as **-o**, **-Ilibrary**, **-S**, etc.). Standard C++ options are ignored by the MPL compiler and passed on to the C++ compiler. The options listed below are read by the MPL compiler.

-LegionL <context path name>

Specifies an additional search path, named in *<context path name>*, for object instantiation. Any number of paths may be specified with this flag, at any place in the compile line.

-LegionOut <output context path>

Tells the compiler that when it registers the class binary in context space it should register it in a specified context path given in *<output context path>*. This path name can be absolute or relative.

-nolegion

Tells the compiler not to register a specified class in context space.

-N

Tells the compiler not to include MPL-specific declarations in the class object file. This allows for multiple MPL files to be separately compiled and linked, avoiding duplicate symbol conflicts.

-trans

Tell the compiler not to delete the trans files (temporary files created by the MPL compiler that store the intermediate C++ representation of MPL binaries) after compilation. This is useful for debugging purposes.

-transname <trans file name>

Tell the compiler to generate trans files called *<trans file name>.c* and *<trans file name>.o* so as to avoid trans files over-writing each other. Note that *<trans file name>* can contain relative or absolute Unix paths with the file name. This flag is most useful for users compiling multiple architectures on a shared file system.

-nocompile

Specify that the compiler should only translate MPL to C++ and not compile the generated C++ code.

-FOxxx

Specify additional options for the MPL compiler to use in generating code.

The *xxx* may be any of the following:

- selfless** - Calls made through the SELF pointer should not be treated as Mentat calls (i.e., as parallel) but as regular C++ calls.
- nopack** - Do not attempt to automatically generate pack or unpack functions on classes.
- mapstdio** Map stdio and stderr from the MPL objects to a tty object. If no tty object is available, output will be lost.

8.0 Unsupported C++ Features

The current compiler does not support the full ANSI C++ definition. In particular we do not support:

- Exceptions
- **typedefs** or **enums** used in Mentat classes
- Pointers passed to or returned from Mentat member functions
- Inheritance of C++ classes or Mentat classes from other Mentat classes
- Virtual functions defined anywhere in an inheritance graph which contains a Mentat class
- Calling a function in a C++ base class on a Mentat class derived from that base class
- Template Mentat classes (note that you can use instantiated templates as member in, parameters to, or return values from Mentat classes)

If you need these features, please let us know.

9.0 Programming Language Points



This is a list of rules that can help the programmer avoid common pitfalls. Most of these points stem from the address-space-disjoint nature of Mentat objects.

- 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.
- 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. If the programmer wants the effect of public member variables, appropriate member functions can be defined.
- Programmers cannot assume that pointers to instances of Mentat classes point to the member data for the instance.

`x = my_pointer->y; // will not work.`

- Mentat classes cannot have any friend classes or functions. 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.
- Parameters are copied *by value*.
- Virtual functions *do not* work as expected between Mentat object address spaces. Virtual functions rely on pointers to the function's definition; such pointers are meaningless in distributed memory systems.
- While overloading of Mentat object constructors is allowed, the generated code is not actually used to construct the object but rather to initialize it. Normally, this is an unimportant distinction but it becomes important when dealing with user-defined pack and unpack functions on Mentat classes. These functions are automatically called by the Mentat object when it is created but before it is initialized. For this reason, the user must take into account the fact that pointers in the class may not be initialized by a constructor.

10.0 References

- [Ada] *Reference Manual for the Ada Programming Language*, United States Department of Defense, Ada Joint Program Office, July 1982.
- C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, pp. 666-677, August, 1978.
- A. S. Grimshaw, J. B. Weissman, and W. T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing," to appear *ACM Transactions on Computer Systems*, July, 1993.
- A. S. Grimshaw, "The Mentat Computation Model - Data-Driven Support for Dynamic Object-Oriented Parallel Processing," *Computer Science Technical Report*, CS-93-30, University of Virginia, May, 1993.
- A. S. Grimshaw, E. A. West, and W.R. Pearson, "No Pain and Gain! - Experiences with Mentat on Biological Application," *Concurrency: Practice & Experience*, pp. 309-328, Vol. 5, issue 4, June, 1993.
- A.S. Grimshaw, J.B.Weissman, E.A. West, and E. Loyot, "Meta Systems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, pp. 257-270, vol. 21, no. 3, June, 1994.
- A. S. Grimshaw, W. T. Strayer, and P. Narayan, "Dynamic Object-Oriented Parallel Processing," *IEEE Parallel & Distributed Technology: Systems & Applications*, pp. 33-47, May, 1993.
- A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, pp. 39-51, May, 1993.
- B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd Edition, 1991.
- J. B. Weissman, A. S. Grimshaw, and R. Ferraro, "Parallel Object-Oriented Computation Applied to a Finite Element Problem," *Scientific Computing*, vol. 2, no. 4, pp. 133-144, Feb., 1994.

Index**A**

accumulators 43
ADA 26

B

blocking 13, 44
bound() 19, 47

C

call-by-value 29
common pitfalls 51
computational complexity 7, 12
contained objects 10
control dependencies 5

D

data dependencies 5, 7
delinearization 29–35

E

encapsulation 7, 8

F

Fibonacci 38, 45
Fortran 42

G

guard 27
guard-statement 27

I

independent objects 10
instance 15
invocation 47

L

latency 12
legacy code 37, 40, 40–42
LegionBuffer 34–35
linearization 29–35
loid() 19
loop-splitting 43–44

M

Mentat class definitions 47
Mentat class interface 11
Mentat member function invocation 47
Mentat object instantiation 15
Mentat object model 10
Mentat objects 12
mentat_handle 17–18, 19, 20
mentat_object 17–18, 21
mentat_return 22–24, 28
MPL 5
mselect/maccept 26–28, 36, 40

N

new() 16
new(...) 47

P

pack/unpack 30–36
parallel implementation 38
parallel recursion 45
parallelism encapsulation 8–9, 13
parameter passing 29–36, 47
problem domain 7

S

same source application 39
same source implementation 38, 40
self 47
sequential implementation 38
sequential object 11, 14, 36
stateful mentat class 13, 23
stateful object 11, 14, 36
stateless mentat class 13
stateless object 11, 18, 46

T

tail recursion 24, 44, 44–45

V

virtual function 51
void() 47