Easy-to-use Object-Oriented Parallel Processing with Mentat

Andrew S. Grimshaw

Technical Report No. CS-92-32

Easy-to-use Object-Oriented Parallel Processing with Mentat

Andrew S. Grimshaw
Department of Computer Science
University of Virginia
Charlottesville, VA, 22903
grimshaw@virginia.edu, 804-982-2204

Abstract

Writing portable applications for parallel architectures has proven to be more difficult than writing sequential software. This is due in large part to the lack of easy-to-use, high-level abstractions. Mentat is a portable object-oriented parallel processing system that extends object encapsulation to include parallelism encapsulation. In Mentat, programmers are responsible for identifying those classes that are of sufficient computational complexity to warrant parallel execution. The compiler and run-time system manage program graph construction, communication, synchronization, and scheduling. Mentat has been implemented on Sun workstations, the Silicon Graphics Iris, the Intel iPSC/2, and the Intel iPSC/860. We present the Mentat programming language, including several examples, the Mentat virtual machine architecture, and performance results from two of the supported architectures.

Keywords: parallel processing, object-oriented, programming languages.

Easy-to-use Object-Oriented Parallel Processing with Mentat

1. Introduction

Two problems plague parallel MIMD architecture programming. First, writing parallel programs by hand is very difficult. The programmer must manage communication, synchronization, and scheduling of tens to thousands of independent processes. The burden of *correctly* managing the environment often overwhelms programmers, and requires a considerable investment of time and energy. Second, once implemented on a particular MIMD architecture, the resulting codes are usually not usable on other MIMD architectures; the tools, techniques, and library facilities used to parallelize the application are specific to a particular platform. Thus, considerable effort must be re-invested to port the application to a new architecture. Given the plethora of new architectures and the rapid obsolescence of existing architectures, this represents a continuing time investment.

Mentat has been developed to directly address the difficulty of programming MIMD architectures and the portability of applications. The three primary design objectives are to provide 1) easy-to-use parallelism, 2) high performance via parallel execution, and 3) applications portability across a wide range of platforms. The premise underlying Mentat is that writing programs for parallel machines does not have to be hard. Instead, it is the lack of appropriate abstractions that has kept parallel architectures difficult to program, and hence, inaccessible to mainstream, production system programmers.

The Mentat approach exploits the object-oriented paradigm (see Appendix A) to provide high-level abstractions that mask the complex aspects of parallel programming, communication, synchronization, and scheduling, from the programmer. Instead of worrying about, and managing, these details, the programmer is free to concentrate on the application. The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution. The complex tasks, scheduling, communication and synchronization, are handled by Mentat.

There are two primary components of Mentat: the Mentat Programming Language (MPL) [9] and the Mentat run-time system [8]. The MPL is an object-oriented programming language based on C++ that masks the complexity of the parallel environment from the programmer. The granule of computation is the Mentat class member function. Mentat classes consist of contained objects (local and member variables), their procedures, and a thread of control. The programmer is responsible for identifying those classes that are of sufficient computational complexity to allow efficient parallel execution. Mentat class instances are used almost exactly like C++ classes.

Mentat extends object encapsulation from implementation and data hiding to include parallelism encapsulation. Parallelism encapsulation takes two forms that we call *intra-object* encapsulation and *inter-object* encapsulation (see Appendix B). Intra-object encapsulation of parallelism means that callers of a Mentat object member function are unaware of whether the implementation of a member function is sequential or parallel, i.e., whether its internal execution graph is a single sequential node, or whether it is parallel. Inter-object encapsulation of parallelism means that 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. Thus, 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 and avoid the weaknesses of each. The underlying assumption is that the programmer can make better decisions regarding program and data partitioning, while the compiler can better manage communication and synchronization. This simplifies the task of writing parallel programs, making parallel architectures more accessible to non-computer scientists.

What makes Mentat different from the dozens of other concurrent and distributed object-oriented systems and languages is its emphasis on parallelism and high-performance. Mentat is not yet another RPC based system, it is a parallel processing system that uses parallel processing compiler and run-time support technology in conjunction with the object-oriented paradigm to produce an easy-to-use high performance system that facilitates hierarchies of parallelism via the mechanism of parallelism encapsulation.

Our objective is to show that Mentat is an easy-to-use, portable, high-performance parallel processing system. We begin by introducing the Mentat programming language (MPL) and the object model. We include several examples that illustrate the use of the language and the resulting parallelism. We next introduce the run-time system that underlies Mentat and provides the virtual machine abstraction that can be easily ported to new architectures. We then proceed to show that we have met our high-performance objectives, by providing performance figures for two applications on two different platforms, a network of Sun workstations and the Intel iPSC/2.

2. The Mentat Programming Language (MPL)

MPL is an extended C++ designed to simplify the task of writing high-performance parallel applications by supporting both intra- and inter-object parallelism encapsulation. This high level ease-of-use objective is realized via four, more specific, design goals.

First and foremost, the MPL is object-oriented. The object-oriented approach to programming has become popular because it promises to both simplify software development and to facilitate software reuse. The object-oriented paradigm is ideal for parallel and distributed systems because users of an object interact with the object via the object's interface. Because of the data hiding, or encapsulation, properties of objects, users cannot directly access private object data. This simplifies concurrency control on object data structures, so objects can be treated as monitors. We extended the usual notions of data and method encapsulation to include *parallelism encapsulation*.

The second design goal contributing to ease-of-use is that the MPL is an extension of an existing language, C++, and minimizes the number of extensions and changes to the base language. The syntax and semantics of the extensions follow the pattern set by the base language, maintaining its basic structure and philosophy whenever possible.

Third, the language constructs have a natural mapping to the macro data flow model, the computation model underlying Mentat. It is a medium grain, data driven model in which programs are directed graphs. The vertices of the program graphs are computation elements (called *actors*) that perform some function. The edges model data dependencies between the actors. The responsibility for performing the mapping is not the programmer's.

Fourth, the concepts used as the basis of the extensions are applicable to a broad class of languages, so that the Mentat approach can be easily used in other contexts.

These goals have been met in the MPL by extending C++ in three ways, by the specification of Mentat classes, the `rtf`() value return mechanism, and the select/accept statement. The basic idea is to allow the programmer to specify those C++ classes that are of sufficient computational complexity to warrant parallel execution and let the compiler manage communication and synchronization between instances of these classes. Instances of Mentat classes are called Mentat objects. The programmer uses Mentat objects just as any other C++ object. 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. During the course of execution a graph node (member function) may itself be transparently implemented in a similar manner by a macro data flow subgraph. Thus we obtain intra-object parallelism encapsulation; the caller only sees the member function invocation.

Throughout this section we develop several examples that illustrate MPL concepts, and show how the parallelism is realized. The discussion assumes a familiarity with C++ terms. See Appendix C for a short primer on **C++**.

2.1 Mentat Class Definition

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 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, structs, 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.

Because Mentat objects are address space disjoint, member function calls are call by value. Results of member functions are also returned by value. Pointers to objects, particularly variable size objects, may be used as both parameters and as return types. To provide the programmer a way to control the degree of parallelism, Mentat allows both standard C++ classes and Mentat classes. By default, a standard C++ class definition defines a standard C++ object.

The programmer defines a Mentat class by using the keyword **mentat** in the class definition. The programmer may further specify whether the class is **persistent** or **regular**. The syntax for Mentat class definitions is:

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

Persistent objects maintain state information between member function invocations, while regular objects do not. Thus, regular object member functions are pure functions. Because they are pure

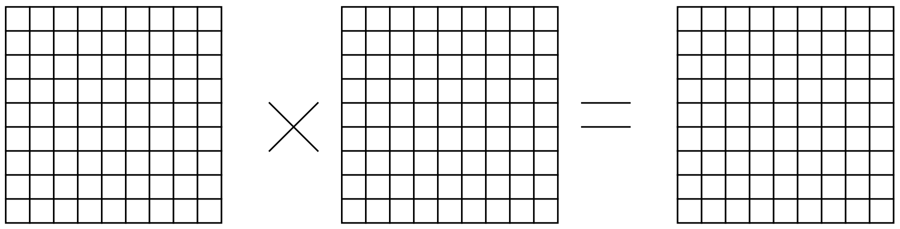
1. The distinction between independent and contained objects is not unusual, and is driven by efficiency considerations.

functions the system is free to instantiate new instances of regular classes at will. Regular classes may have local variables much as procedures do, and may maintain state information for the duration of a function invocation.

When should a class be a Mentat class? In three cases: when its member functions are computationally expensive, when its member functions exhibit high latency (e.g., IO), and when it holds state information that needs to be shared by many other objects (e.g., shared queues, databases, physical devices). 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. Shared state objects should be Mentat classes for two reasons. First, since there is no shared memory in our model, shared state can only be realized using a Mentat object with which other objects can communicate. Second, because Mentat objects service a single member function at a time, they provide a monitor-like synchronization, providing synchronized access to their state.

To illustrate the difference between regular and persistent mentat classes, suppose we wish to perform matrix operations in parallel, e.g., a matrix-matrix multiply. Recall that in a matrix-matrix multiply a new matrix is formed. Each element in the result is found by performing a dot product on the appropriate rows and columns of the input matrices (Figure 2-(a)). Because matrix-matrix multiply is a pure function, we could choose to define a regular mentat class matrix_operators as in Figure 2-(b). 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 and the arguments being transported to them.

Alternatively, we could choose to define a persistent mentat class `p_matrix` as in Figure 2-(c). To use a `p_matrix`, an instance must first be created and initialized with a `matrix*`. Matrix-matrix multiplication can now be accomplished by calling `mpy()`. When `mpy()` is used the argument matrix is transported to the already existing object. Successive calls result in the argument matrices being transported to the same object. In both the persistent and regular case the



(a) matrix-matrix multiplication

```
regular mentat class matrix_operators {
// private members
public:
   matrix* mpy(matrix*,matrix*);
};
```

(b) regular mentat class definition

```
persistent mentat class p_matrix {
// private members
public:
   void initialize(matrix*);
   matrix* mpy(matrix*);
};
```

(c) persistent mentat class definition

Figure 2. Regular versus. persistent classes to perform matrix-matrix multiplication.

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.

2.2 Mentat Object Instantiation & Destruction

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 Mentat objects is slightly different from standard **C++** object instantiation semantics. First, consider the **C++** fragment:

```
{// A new scope
   int X;
   p_matrix mat1;
   matrix_operators m_ops;
} // 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 `p_matrix` is a Mentat class, `mat1` is a *name* of a Mentat object of type `p_matrix`. It is not the instance itself. Thus, `mat1` is analogous to a pointer.

Names (e.g., `mat1`) can be in one of two states, *bound* or *unbound*. An unbound name refers to any instance of the appropriate Mentat class. A bound name refers to a specific instance with a unique name. When an instance of a Mentat class (a Mentat variable) comes into scope or is allocated on the heap, it is initially an unbound name: it does not refer to any particular instance of the class. Thus, a new `p_matrix` is not instantiated when `mat1` comes into scope. When unbound names are used for regular Mentat classes (e.g., `m_ops`), the underlying system logically *creates a new instance for each invocation* of a member function. This can lead to high levels of parallelism, as we shall see later.

2.2.1 Binding and Instantiation

The programmer binds Mentat variables to Mentat objects using four new reserved member functions for all Mentat class objects: `create`(), `bind`(), `bound`(), and `destroy`(). There are three ways a Mentat variable (e.g., `mat1`) may become bound: it may be explicitly created using `create`(), it may be bound by the system to an existing instance using `bind`(), or the name may be assigned to a bound name by an assignment. The `bound`() function indicates whether the mentat object is bound to a particular instance. The member function `destroy`() destroys the named persistent Mentat object. If the name is unbound, the call is ignored.

The `create`() call tells the system to instantiate a new instance of the appropriate class. There are five flavors of `create`(). See Figure 3. Assume the definition `p_matrix mat1`;

```
(a)         mat1.create();
(b)         mat1.create(COLOCATE another_object);
(c)         mat1.create(DISJOINT object1, object2);
(d)         mat1.create(HIGH_COMPUTATION_RATIO);
(e)         mat1.create(int on_host);
(f)         mat1 = expression;
(g)         mat1.bind(THIS_HOST);
```

Figure 3. Binding Mentat variables to Mentat objects.

When `create`() is used as in Figure 3(a), the system will choose on which processor to instantiate the object [8]. The programmer may optionally provide location hints. The hints are COLOCATE, DISJOINT, and HIGH_COMPUTATION_RATIO. These hints allow the programmer to specify where he wants the new object to be instantiated. In Figure 3(b), the programmer has specified that the new Mentat object should be placed on the same processor as the object `another_object`. In Figure 3(c), the programmer has specified that the new object should not be placed on the same processor as any of a list of Mentat objects. In Figure 3(d), the programmer has specified that the new object will have a very high ratio of computation to communication, and thus may be placed on a processor with which it is expensive to communicate. In Figure 3(e), the programmer has specified that the new object should be placed on a specific processor. Names may also be bound as the result of assignment to an expression, as in 3(f).

Mentat variables may also be bound to an already existing instance using the `bind(int scope)` member function, as shown in Figure 3(g). The integer parameter scope can take any one of three values, BIND_LOCAL, BIND_CLUSTER, and BIND_GLOBAL, to restrict the search for an instance to the local host (the host may be a multiprocessor), to the cluster (subnet), and to the entire system respectively.

2.3 Mentat Object Member Function Invocation

Member function invocation on Mentat objects is syntactically the same as for C++ objects. Semantically there are two important differences. Mentat member functions are always call-by-value, and Mentat member function invocation is non-blocking. The non-blocking nature of Mentat object member functions provides for the parallel execution of member functions whenever data dependencies permit. This is transparent to the user and is called inter-object parallelism encapsulation.

As noted earlier, because Mentat objects are address space disjoint, Mentat class member functions always use call-by-value semantics. When pointers are used as arguments, the object (or structure) to which the pointer points is sent to the callee. If the object to which the pointer points is of variable size, then the class of the object must provide a member function `int size_of`() that returns the size of the object in bytes. If a structure or class has contained pointers, they are not “chased”. Call-by-value semantics is common in systems that provide an RPC-like service. The alternative is to allow pointer passing between address spaces.

Example 2. Consider the code fragment shown in Figure 4. The member function `open`() takes two parameters and returns an integer. The first parameter is of type `string*`. Because strings are of variable length, we have provided the function int `size_of()`. Size_of is called at run-time to determine the size of the first parameter, and `size_of()` bytes will be sent to the Mentat object `f`. The second argument is an integer. Fixed size arguments such as integers and structs do not require a `size_of()` function. The compiler ensures that the correct amount of data is transferred.

```
class string {
public:
   int size_of();
}
int string::size_of() {return(strlen(this)+1)};
persistent mentat class m_file {
public:
   int open(string* name, int mode);
   data_block* read(int offset, int num_bytes);
   void write(int offset, int num_bytes, data_block* data);}
{
   // *** A code fragment using m_file
   m_file f;
   f.create(); // No location hints.
   int x = f.open((string*) "my_file",1);
   if (x < 0) {/* error code */}
}
```

Figure 4. Class m_file declaration and use.

The example in Figure 4 illustrates the creation of a persistent object and a simple RPC to a member function of that object. The difference between Mentat and a traditional RPC is what happens when an RPC call is encountered. In traditional RPC, the arguments are marshalled (packaged into a message), sent to the callee, and the caller blocks waiting for the result. The callee accepts the call, performs the desired service, and returns the results to the caller. The caller then unblocks and proceeds.

In Mentat, when a Mentat object member function is encountered, the arguments are marshalled and sent to the callee, but the caller does not block waiting for the result (`x` in Figure 4). Instead, the run-time system monitors (with code provided by the compiler) where `x` is used. If `x` is later used as an argument to a second or third Mentat object invocation, then arrangements are made to send `x` directly to the second and third member function invocations. If `x` is used locally in a strict operation, e.g., `y=x+1;`, or `if (x<0)`, then the run-time system will automatically block the caller and wait for the value of `x` to be computed and returned. This is the case in Example 2. Note, though, that if `x` is never used locally (except as an argument to a Mentat object member function) then the caller never blocks and waits for `x`. Indeed `x` might never be sent to the caller, it might only be sent to the Mentat object member functions for which it is a parameter. This is illustrated in Example 3.

Example 3. This example illustrates construction of a simple pipeline process. We define the Mentat class `data_processor`. The member functions `filter_one`() and `filter_-two`() are filters that process blocks of data. Consider the code fragment in Figure 5. After some

```
regular mentat class data_processor {
public:
   data_block* filter_one(data_block*);
   data_block* filter_two(data_block*);
}


 m_file in_file,out_file;
 data_processor dp;
 in_file.create();out_file.create();
 int i,x; x = in_file.open((string*)"input_file",1);
 x = out_file.open((string*)"output_file",3);
 data_block *res;
 for (i=0;i<MAX_BLOCKS,i++) {
   res = in_file.read_block(i);
   res = dp.filter_one(res);
   res = dp.filter_two(res);
   out_file.write_block((i*BLK_SIZE,res);
}
```

Figure 5. A Pipelined Data Processor. The main loop reads MAX_BLOCKS data_blocks, passes them through filter_one() and filter_two(), and then writes them to the output file. The loop is unrolled at run-time and a pipeline (Figure 6) is formed.

initialization, creating and opening input and output files, the loop in the code fragment sequentially reads MAX_BLOCKS data blocks from the file "input_file", processes them through filters one and two, and writes them to "output_file". Note that the variable `res` is used as a temporary variable and as a conduit through which information passes between the filters. This fragment is written in a manner that is natural to C programmers.

In a traditional RPC system, this fragment would execute sequentially. Suppose that each member function execution takes 10 time units, and that each communication takes 5 time units. Then the time required to execute an iteration of the loop in a sequential RPC system is the sum of four times the member function execution time, plus seven times the communication time (because all parameters and results must be communicated from/to the caller). Thus the total time required is 75 time units.

The average time per iteration for the Mentat version is considerably less, just over 10 time units. First observe that the time for a single iteration is four times the communication time, 20 time units, plus four times the execution time, 40 time units, for a total of 60 time units. There are only four communications because intermediate results are not returned to the caller, rather they are passed directly where needed. Next, consider that the reads, the two filter operations, and the writes can be executed in a pipelined fashion with each operation executing on a separate processor (see Figure 6).

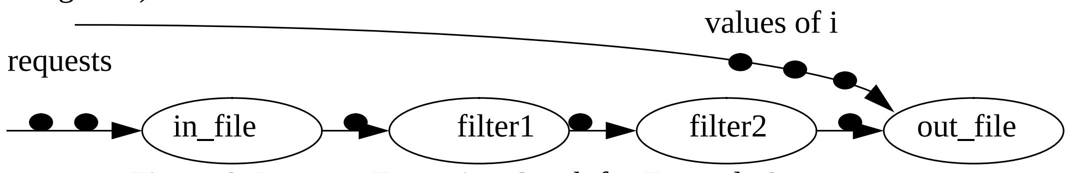


Figure 6. Program Execution Graph for Example 3.

Under these circumstances, each of the four member function invocations, and all of the communication, can be performed concurrently. The communication for the i^{th} iteration can be overlapped with the computation of the $(i+1)^{th}$ iteration. (We assume that communication is asynchronous and that sufficient communication resources exist.)

Using a standard pipe equation

T_{All} = time for all iterations
T_{Stage} = time for longest stage = 10 time units
T_1 = time for first iteration = 60 time units
T_{Avg} = average time per iteration
$T_{All} = T_1 + T_{Stage} * (MAX\_BLOCKS-1)$
$T_{All} = 60 + 10*(MAX\_BLOCKS-1)$
$T_{Avg} = (60 + 10*(MAX\_BLOCKS-1))/MAX\_BLOCKS$

When MAX_BLOCKS is one, the time to complete is 60 time units, with an average of 60 time units. This is marginally faster than a pure RPC (75 time units) because we don't send intermediate results to the caller. When MAX_BLOCKS is greater than one, the time required for the first iteration is 60 time units, and successive results are available every ten time units. Thus, as MAX_BLOCKS increases, the average time per iteration drops, and approaches 10 time units.

Now consider the effect of quadrupling the time to execute filter one from 10 to 40 time units. The time to execute the traditional RPC version goes from 75 to 105 time units. Using the standard pipe equation the first result is available at time 90, and successive values every 24 time units. The standard pipe equation assumes that there is just one functional unit for each stage. This assumption is invalid in Mentat in this example, and the time per iteration for the Mentat version remains unchanged at 10 time units, if there are sufficient computation resources. To see why, consider that the `data_processor` class is a regular Mentat class. This means that the system may instantiate new instances at will to meet demand. A new instance of data_processor to service filter_one requests is created whenever a result is generated by the read. There would be five instances of the `data_processor` class active at a time, four performing filter one, and one performing filter two.

There are four items to note from this example. First, the main loop may have executed to completion (all `MAX_BLOCKS` iterations) before the first write has completed! Second, suppose our "caller" (the main loop) was itself a server servicing requests for clients. Once the main loop is complete the caller may begin servicing other requests while the first request is still being completed. Third, the order of execution of the different stages of the different iterations can vary from a straight sequential ordering, e.g., the last iteration may "complete" before earlier iterations. This can happen, for example, if the different iterations require different amounts of filter processing. This additional asynchrony is possible because the run-time system guarantees that all parameters for all invocations are correctly matched, and that member functions receive the correct arguments. The additional asynchrony permits additional concurrency in those cases where execution in strict order would prevent later iterations from executing even when all of their synchronization and data criteria have been met. Finally, in addition to the automatic detection of inter-object parallelism opportunities, we may also have intra-object parallelism encapsulation, where each of the invoked member functions may be internally parallel. Thus we obtain even more parallelism.

2.4 The Return-to-Future Mechanism

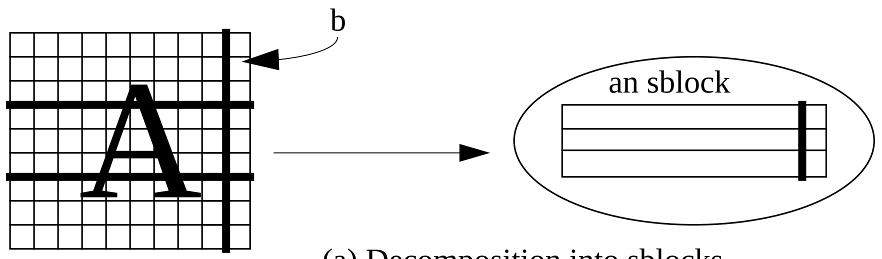
The return-to-future function (`rtf`()) is the Mentat analog to the `return` of **C**. Its purpose is to allow Mentat member functions to return a value to the successor nodes in the macro data-flow graph in which the member function appears. Mentat member functions use the `rtf`() as the mechanism for returning values. 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.

While there are many similarities between `return` and `rtf`(), `rtf`() differs from a `return` in three significant ways.

First, in **C**, before a function can `return` a value, the value must be available. This is *not* the case with an `rtf`(). Recall that when a Mentat object member function is invoked, the caller does not block, rather we ensure that the results are forwarded wherever they are needed. Thus, a member function may `rtf`() a "value" that is the result of another Mentat object member function that has not yet been completed, or perhaps even begun execution. Indeed, the result may be computed by a parallel subgraph obtained by detecting inter-object parallelism.

Second, a **C** `return` signifies the end of the computation in a function, while an `rtf`() does not. An `rtf`() indicates only that the result is available. Since each Mentat object has its own thread of control, additional computation may be performed after the `rtf`(), e.g., to update state information or to communicate with other objects. In the message passing community this is often called send-ahead. By making the result available as soon as possible we permit data dependent computations to proceed concurrently with the local computation that follows the `rtf`().

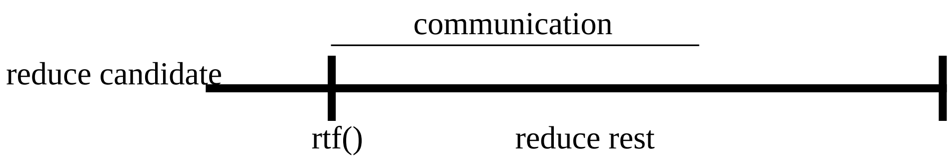
Third, a `return` returns data to the caller. `Rtf`() may or may not return data to the caller



(a) Decomposition into sblocks.

```
vector*sblock::reduce(vector* pivot) {
   reduce current column using pivot
   find candidate row, it has the largest absolute value in current column
   reduce candidate row
   rtf(candidate row);
   reduce remaining rows - this is the computationally expensive part
}
```

(b) sblock::reduce() pseudo-code.



(c) Overlap of communication and computation with rtf().

Figure 7. Gaussian elimination with partial pivoting illustrating the use of rtf() to overlap communication and computation.

depending on the data dependencies of the program. If the caller does not use the result locally, then the caller does not receive a copy. This saves on communication overhead. The next two examples illustrate these features.

Example 4. Consider a `persistent class sblock` used in Gaussian elimination with partial pivoting. In this problem, illustrated in Figure 7, we are trying to solve for x in Ax=b. The `sblocks` contain portions of the total system to be solved. The sblock member function

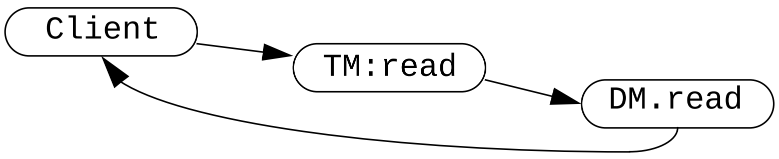
```
vector* sblock::reduce(vector*);
```

performs row reduction operations on a submatrix and returns a candidate row. Pseudo-code for the reduce operation is given in Figure 7-(b). The return value can be quickly computed and returned via rtf(). The remaining updates to the sblock can then occur in parallel with the communication of the result (Figure 7-(c)). In general, best performance is realized when the `rtf`() is used as soon as possible.

Example 5. Consider a transaction manager (TM) that receives requests for reads and writes, and checks to see if the operation is permitted. If it is, the TM performs the operation via the data manager (DM) and returns the result. Figure 8-(a) illustrates how the read operation might be implemented. In a RPC system, the record read would first be returned to the TM, and

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

(a) Code fragment for Transaction Manager - read.



(b) Call graph illustrating communication for TM:read. Arcs represent message traffic.

Figure 8. Tail-recursion in the MPL.

then to the user. In MPL the result is returned directly to the user, bypassing the TM (Figure 8-b). Further, the TM may immediately 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.

2.5 The MPL Compiler - mplc

The mplc is responsible for mapping MPL programs to the macro data-flow model. It accomplishes this by translating MPL programs to C++ programs with embedded calls to the Mentat run-time system. These C++ programs are, in turn, compiled by the host C++ compiler (see Figure 9). This is an approach similar to that used by the AT&T C++ compiler in which C++

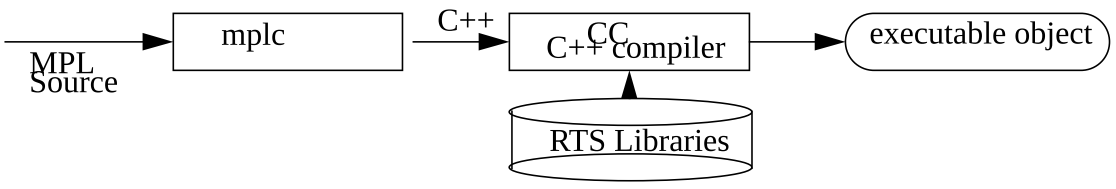


Figure 9. MPL compiler steps.

programs are translated into a portable assembly language, **C**.

2.6 MPL Summary

The MPL is an extension of C++. MPL provides high-level abstractions that allow the programmer to express the decomposition of a program using a natural extension of the object-oriented paradigm. The compiler and run-time system are responsible for mapping this high-level user's view of the program onto the physical hardware.

3. The Run-Time System

The Mentat run-time system [8] supports Mentat programs via the provision of a virtual macro-data flow machine (Figure 10). The virtual machine provides support routines that perform run-time data dependence detection, program graph construction, program graph execution, token matching, scheduling, communication, and synchronization. The compiler generates code that communicates with the run-time system to correctly manage program execution.

The Mentat run-time system is not an operating system. Instead, the run-time system is layered on top of an existing host operating system, using the host operating system's processes, memory, **C** library, and interprocess communication (IPC) services.

The virtual machine model permits the rapid transfer of Mentat to new architectural platforms. Only the machine-specific components need to be modified. Because the compiler uses a virtual machine model, porting applications to a new architecture does not *require* any user source level changes[2]. Once the virtual machine has been ported, user applications are recompiled and can execute immediately.

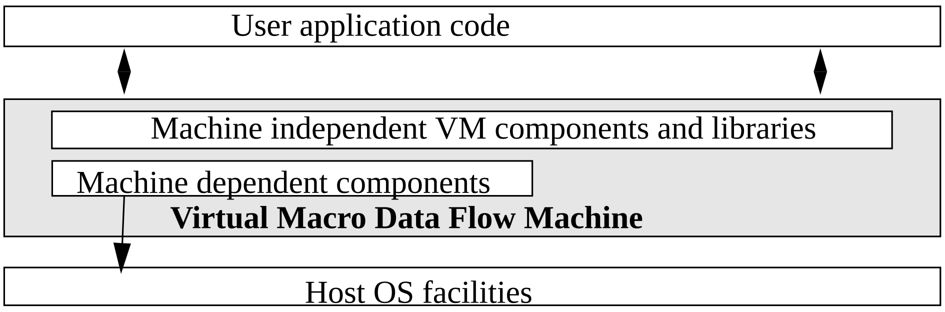
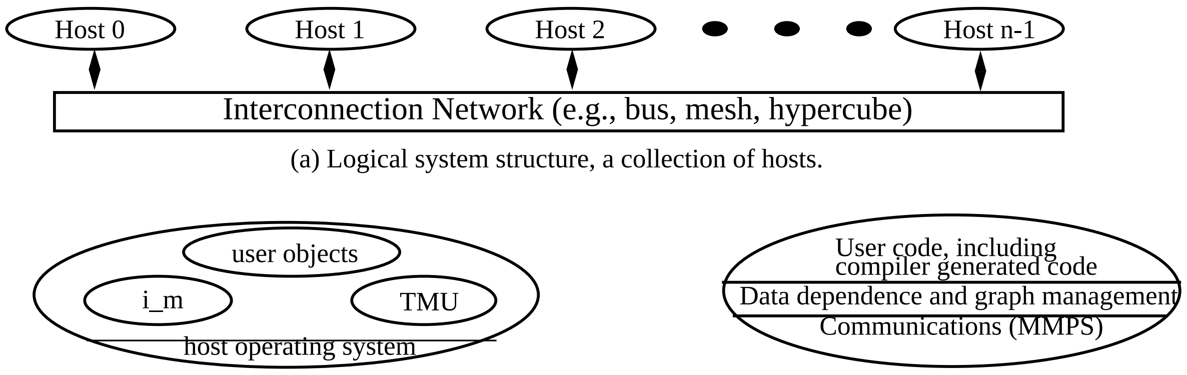

User application code
Machine independent VM components and libraries
Machine dependent components
Virtual Macro Data Flow Machine
Host OS facilities

Figure 10. Mentat Virtual Machine Model.

2. Application code may benefit from changes, but does not *require* them, e.g., on the Sun 3/60 loop unrolling provides no benefit; on the SparcStation it does.

The Mentat virtual macro data flow machine is implemented by the Mentat run-time system (RTS). The RTS is in two sections, run-time libraries that are linked into Mentat objects, and run-time objects that provide run-time services, e.g., scheduling, naming, and binding. The run-time libraries are responsible for program graph construction, select/accept execution, and reliable communication.

The logical structure of a Mentat system is that of a collection of hosts communicating through an interconnection network (see Figure 11). Each host is able to communicate with any other host via the interconnection network, although not necessarily at uniform cost.


Host 0
Host 1
Host 2
Host n-1
Interconnection Network (e.g., bus, mesh, hypercube)
user objects
i_m
TMU
host operating system
User code, including compiler generated code
Data dependence and graph management
Communications (MMPS)

(a) Logical system structure, a collection of hosts.

(b) Logical host structure, user objects and system objects.

(c) Mentat object structure

Figure 11. Mentat run-time system structure.

The logical interconnection network is provided by the lowest layer of the run-time system, **MMPS**. MMPS (Modular Message Passing System) provides an extensible point-to-point message service that reliably delivers messages of arbitrary size from one process to another.

Each host has a complete copy of the run-time system server objects (Figure 11-(b)). These include the *instantiation manager* (*i_m*) and the token matching unit (*TMU*). The instantiation manager is responsible for high level Mentat object scheduling (deciding on which host to locate an object), and for instantiating new instances. The high level scheduling algorithm is distributed, adaptive, and stable. The TMU is responsible for matching tokens for regular objects and instantiating new instances (via the *i_m*) when needed.

Dynamic data dependence detection and program graph construction are accomplished by the mplc in conjunction with the run-time system (Figure 11-(c)). The mplc generates library calls that tell the RTS when certain variables, called potential result variables (PRV's), e.g., `x` in `x=matrix_op.mpy(B,C)` of Example 1, are used on either the left-hand side or the right-hand side of expressions. By carefully observing where PRV's are used at run-time, the RTS can construct data dependency program graphs, and manage communication and synchronization.

One final note on the run-time system: because we use a layered approach and mask differences in the underlying operating system and inter-process communication, applications are completely source code portable between supported architectures. We routinely develop and debug software on Sun workstations and use the sources unchanged on the Intel iPSC/2. In this day of incompatible parallel computers this is quite useful. The fact that the sources are identical allows us to compare architectures using the exact same code, and to measure the effect of known architectural differences on algorithm performance, e.g., to measure algorithm sensitivity to communication latency.

The only real difficulty when porting applications is grain size selection. Each platform has different optimum grain size. To date we have overcome this problem either by decomposing the problem with the largest grain size needed for any platform, or by parameterizing the Mentat class to indicate the number of pieces into which the problem should be decomposed. We are currently examining ways to automate this process based on information provided by the programmer.

4. Performance

Ease-of-use and programming models aside, the bottom line for parallel processing systems is performance. As of this writing we have implemented the Mentat run-time system and run benchmarks on a network of Sun workstations (3's and 4's), the Silicon Graphics Iris, the Intel iPSC/2, and the Intel iPSC/860. Speedups for two benchmarks on the Sun 3 and the iPSC/2 are given below. In each case the speedup shown is relative to an equivalent **C** program, **not** relative to the Mentat implementation running on one processor. We have been very careful to use the same level of hand optimization of inner loops, and the same level of compiler optimization for both the **C** and MPL versions. In both cases the inner loops were optimized in **C** using standard loop optimization techniques; no assembly language was used. The two benchmarks are matrix multiply and Gaussian elimination. Each benchmark was executed for several matrix dimensions, e.g., 100×100, 200×200. Single precision (32 bit) values were used. Matrix multiply and Gaussian elimination were chosen because they are *de facto* parallel processing benchmarks. All times are carriage return to complete times, i.e., all overhead including loading of object executables and data distribution have been included.

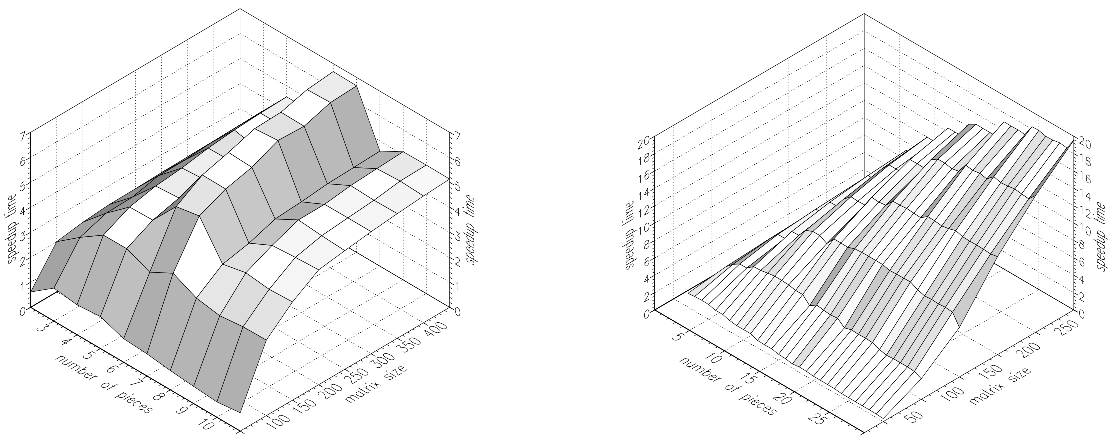
4.1 Execution Environment

The network of Suns consists of 8 Sun 3/60's serviced by a Sun 3/280 file server running NFS connected by thin Ethernet. All of the workstations have eight megabytes of memory and a MC68881 floating point coprocessor.

The Intel iPSC/2 is configured with thirty-two nodes. Each node has four megabytes of physical memory and an 80387 math co-processor. The nodes are not equipped with either the VX vector processor or the SX scalar processor. The NX/2 operating system provided with the iPSC/2 does not support virtual memory. The lack of virtual memory, coupled with the amount of memory consumed by the operating system, limited the problem sizes we could run on the iPSC/2.

Matrix Multiply

The implementation tested is for the regular Mentat class matrix_operators. The Mentat times include the time to copy the arguments. The speed-ups for matrix multiply are shown in Figures 13-(a) and 13-(b) below. The algorithm (and application source) is the same for both systems. Suppose the matrices *A* and *B* are to be multiplied. If k pieces are to be used, the B matrix is split into sqrt(k) vertical slices, and the A matrix into k/sqrt(k) horizontal slices. Each of the sqrt(k)*(k/sqrt(k)) workers gets an appropriate piece of A and of B to multiply. The results of the invocations are merged together and sent to computations that are dependent on the result of the *A*B* operation. The effect of the partitioning can be clearly seen in 13-(a). The speedup for four



(a) 8-processor Sun 3/60 network (b) 32-processor Intel iPSC/2

Figure 13. Speedup for matrix multiply.

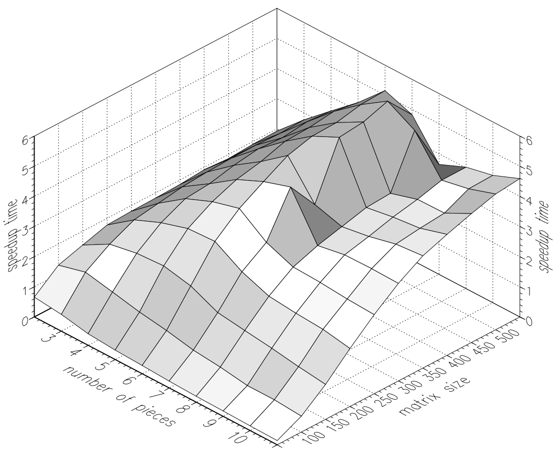
pieces is the same as five pieces, and the speedup for six pieces is the same as for seven pieces. This is because the underlying object will not split the work into five or seven pieces. The fall off in 13-(a) at eight pieces is due to the fact that there are only eight processors, as a result of which the scheduler must place two objects on one processor.

Gaussian Elimination

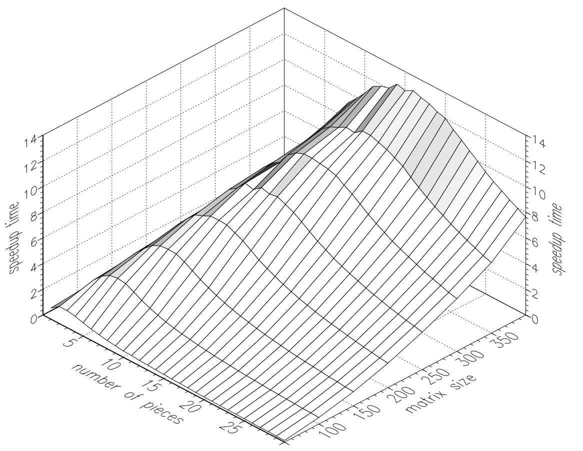
In our algorithm, the controlling object partitions the matrix into *n* strips and places each strip into an instance of an *sblock*, a Mentat class. Then, for each row, the reduce operator is called for each *sblock* using the partial pivot calculated at the end of the last iteration. The reduce operation of the *sblock* reduces the *sblock* by the vector, selects a new candidate partial pivot, and forwards the candidate row to the controlling object for use in the next iteration. This algorithm results in frequent communication and synchronization. The effect of frequent synchronization can be clearly seen when the speed-up results for Gaussian elimination in Figures 14-(a) and 14-(b) are compared to the results for matrix multiply.

5. Related Work

Mentat does not exist in a vacuum. There are many other systems and projects that are similar in some respects to Mentat, and that share many of the same goals. Mentat has much in common with both distributed object-oriented systems and with parallel processing systems.



(a) 8-processor Sun 3/60 network



(b) 32-processor Intel iPSC/2

Figure 14. Speedup for Gaussian elimination.

Mentat inherits features from both, ease of use from the object-oriented paradigm, and models and compiler techniques from the parallel processing domain.

What distinguishes Mentat from other distributed object-oriented systems is the combination of its objectives: easy-to-use high performance via parallelism, and reliance on compiler and run-time techniques to transparently exploit parallelism at multiple levels by managing inter-object data dependence. Many other systems [1,7] have fault-tolerance (e.g., transaction support), the use of functional specialization (e.g, file servers), and the support of inherently distributed applications (e.g., email), as their primary objectives. For most of these systems high-performance is simply not an issue, leading the implementations to rely on blocking RPC-like mechanisms, as opposed to the non-blocking invocations of Mentat. Alternatively, many systems permit parallelism, but require the programmer to exploit and manage it.

In the object-oriented parallel processing domain Mentat differs from systems such as [2,4] (shared memory C++ systems) in its ability to easily support both shared memory MIMD and distributed memory MIMD architectures, as well as hybrids. PC++ [10] on the other hand, is a data parallel C++. Mentat accommodates both functional and data parallelism, often within the same program. ESP [12] is perhaps the most similar of the parallel object-oriented systems. It too is a high-performance extension to C++ that supports both functional and data parallelism. What distinguishes Mentat is our compiler support. In ESP remote invocations either return values or futures. If a value is returned, then a blocking RPC is performed. If a future is returned, it must be treated differently. Futures may not be passed to other remote invocations, limiting the amount of parallelism. Finally, ESP supports only fixed size arguments (except strings). This makes the construction of general purpose library classes, e.g., matrix operators, difficult.

In the parallel processing (as opposed to distributed systems) domain there exists a spectrum of languages for writing parallel applications, from fully explicit manual approaches to implicit compiler-based approaches. In fully explicit approaches a traditional language such as C or Fortran is extended with communication and synchronization primitives such as send and receive or shared memory and semaphores. The advantages of this approach are that 1) it is relatively easy to implement, 2) it reflects the underlying hardware model, and 3) the programmer can use application domain knowledge to optimally partition and schedule the problem. However, the programmer must also correctly manage communication and synchronization. This can be a difficult task, and can overwhelm the programmer, particularly in the presence of “Heisenbugs”[3]. Low-level primitives such as send and receive are the assembly language of parallelism. Anything can be done with them, but at the cost of increased burden on the programmer. Therefore, much as high-level languages and compilers were developed to simplify sequential programming, compilers have been built for parallel systems.

In fully automatic compiler-based approaches the compiler is responsible for performing dependence analysis and finding and exploiting opportunities for parallelism [11]. Compiler-based approaches are usually applied to Fortran. Ideally, application of this approach would permit the automatic parallelization of “dusty deck” Fortran programs. The advantage of compiler-based techniques is that the compiler can be trusted to get communication and synchronization right. The problem is that compilers are best at finding fine-grain and loop level parallelism, and not good at detecting large-grain parallelism. This is because they lack knowledge of the applica-

3. Heisenbugs are timing dependent bugs, in particular, those bugs that go away when debugging, or tracing, is turned on. They are among the most frustrating to find.

tion, forcing them to “reason” about the program using a fine-grain dependence graph. Message passing MIMD architectures require medium-to coarse-grain parallelism in order to operate efficiently. Thus, purely compiler-based approaches are inappropriate for this class of machines because of the mismatch of granularity. Recently, there have been attempts to exploit programmer knowledge to improve data distribution [5]. This approach is best suited to data parallel problems.

Mentat strikes a balance that captures the best aspects of both explicit and compiler-based approaches. The user makes granularity and partitioning decisions using high-level Mentat class definitions, while the compiler and run-time system manage communication, synchronization, and scheduling. We believe that such hybrid approaches offer the best ease-of-use/performance trade-off available today. In the long term, we expect compiler technology to improve, and the need for programmer intervention to decrease.

With respect to applications portability, there have been several mechanisms introduced recently that provide for application portability across platforms. Examples include PVM [3], and Linda [6]. They, like Mentat, achieve portability by providing a virtual machine interface to the programmer. The virtual machine can then be ported to new architectures, and if the applications programmer is limited to that interface, the application will port.[4]

The key difference between Mentat and these systems is the level at which applications must be written. Other systems are low-level, explicit parallel systems, suffering all of the disadvantages, and gaining all of the advantages, of fully explicit systems.Mentat provides a high-level language, eliminating the disadvantages.

6. Summary

Writing software for parallel and distributed systems that makes effective use of the available CPU resources has proven to be more difficult that writing software for sequential machines.

4. Virtual machine abstractions are nothing new. This concept was carried to its logical extreme in the 1970’s in the UCSD P-machine. There existed p-machine implementations for every major microprocessor of the day, 6502, 6809, Z-80, 8080, and the 8086. Programs were object-code compatible between supported architectures. Only one executable was ever needed.

This is true even though most of the work has been done by programmers that have a good understanding of the machines on which they're working. Given the current software crisis for sequential machines, it is unlikely that parallel architectures will be widely used until software tools are available that hide the complexity of the parallel environment from the programmer.

We have presented the Mentat approach to solving the parallel software problem. The key idea is to have programmers use application domain knowledge to decompose the problem, and to use compiler technology to correctly manage scheduling, communication, and synchronization. By exploiting the strengths of both programmers (domain knowledge) and the compiler (correctness) we can leverage the programmers' efforts, freeing them from detail, and increasing their productivity.

Our approach includes extending the object-oriented paradigm's encapsulation techniques to include parallelism encapsulation. Programmers use domain knowledge to specify those classes that are computationally complex. We then apply compiler technology to encapsulate the parallelism internal to an object, and to exploit parallelism opportunities between objects.

It is possible, indeed probable, that a good programmer could write a more efficient and concurrent program using raw send and receive. We believe, though, that send and receive (and semaphores) are the assembly language of parallelism. Just as early high level language compilers could be beaten by a good programmer writing in assembly language, MPL performance can be beaten by a hand coded application using send and receive. Extending the analogy, just as high level languages now have good optimizing compilers that do as well as most programmers, and better than many, we expect MPL compiler technology to improve. Indeed, several optimizations are already planned. The question that must be answered for both high level languages versus assembly languages and for MPL versus raw send and receive is whether the simplicity and ease of use are worth the performance penalty. We believe that they are.

7. References

[1] H. Bal, J. Steiner, and A. Tanenbaum, “Programming Languages for Distributed Computing Systems,” *ACM Computing Surveys*, pp. 261-322, vol. 21, no. 3, Sept. 1989.

[2] B. Beck, “Shared Memory Parallel Programming in C++,” *IEEE Software*, 7(4) pp. 38-48, July, 1990.

[3] A. Beguelin et al.,”A Users’ Guide to PVM (Parallel Virtual Machine)”, Oak Ridge National Laboratory TM-11826.

[4] B. N. Bershad, E. D. Lazowska, and H. M. Levy, “Presto: A System for Object-Oriented Parallel Programming,” *Software - Practice and Experience*, 18(8), pp. 713-732, August, 1988.

[5] D. Callahan and K. Kennedy,”Compiling Programs for Distributed-Memory Multiprocessors” *The Journal of Supercomputing*, no. 2, pp. 151-169, 1988, Kluwer Academic Publishers.

[6] N. Carriero and D. Gelernter, “Linda in Context,” *Comm. of the ACM*, pp. 444-458, April, 1989.

[7] R. Chin and S. Chanson, “Distributed Object-Based Programming Systems,” *ACM Computing Surveys*, pp. 323-358, vol. 21, no. 3, Sept. 1989.

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

[9] A. S. Grimshaw, E. Loyot Jr., and J. Weissman, “Mentat Programming Language (MPL) Reference Manual,” University of Virginia, Computer Science TR 91-32, 1991.

[10] J. K. Lee and D. Gannon, “Object Oriented Parallel Programming Experiments and Results,” *Proceedings of Supercomputing ‘91*, pp. 273-282, Albuquerque, NM, 1991.

[11] C. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[12] S. K. Smith, et al., “Experimental Systems Project at MCC,” MCC Technical Report Number: ACA-ESP-089-89, March 2, 1989.

Appendix A

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 can compilers. 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.

These two observations lead to our underlying philosophy; exploit the comparative advantages of both humans and compilers. Humans understand the problem domain and can best make partitioning decisions, while compilers “understand” data dependence and scheduling. Therefore, in Mentat, programmers tell the compiler using a few key words what computations are worth doing in parallel and what data are associated with the computations. The compiler then takes over and does what it does best, manage parallelism.

Appendix B

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_op` 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.

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 multiplies, B*C and D*E, are not data dependent on one another and can be safely executed in parallel. 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 execution graph is shown in Figure 1-a.

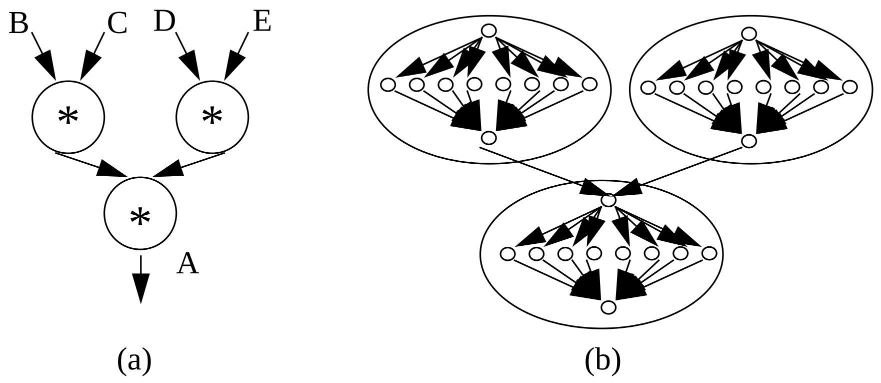


Figure 1. Parallel Execution of Matrix Multiply Operations.
(a) Inter-object parallelism encapsulation.
(b) Intra-object parallelism encapsulation where the multiplies of (a) have been transparently expanded into parallel subgraphs.

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 multiply. Suppose the matrix multiplies are themselves executed in parallel (with the parallelism detected in a manner similar to the above). Further, suppose that each multiply is executed in eight pieces (Figure 1-b). 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 multiply 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!

Appendix C

A Brief Introduction to C++

C++ is an object-oriented extension of C developed by Bjarne Stroustrup of AT&T Bell Labs that avoids the performance penalty usually associated with object-oriented languages. C++ supports object-oriented concepts such as objects, classes, encapsulation, inheritance, polymorphism, and function and operator overloading. The most important extensions revolve around *classes*. Classes are structurally similar to C structs, and in many implementations are implemented as structs.

Classes support the concept of encapsulation via the provision of *private* and *protected member variables* and *member functions*. Private members (e.g., `top`) may only be accessed by member functions of the class (e.g., `push()`). Protected members may be accessed only by members of the class and be members of derived classes. Non-accessible members may still be indirectly manipulated via public member functions. By limiting access to members, the language provides support for encapsulation.

Inheritance means that classes may be defined in terms of other classes, inheriting their behavior (public, private, and protected members). When a class may have at most one super (parent) class, we say a language supports *single inheritance*, and a tree-like class structure results.When there may be multiple super classes we say the language supports *multiple inheritance*. C++ supports multiple inheritance.

A limited form of polymorphism is supported in C++ via *virtual functions*. Multiple classes, all derived from the same base class, may all define different implementations of a virtual function. When the function is invoked on a pointer or reference to an instance of the base class the appropriate function is bound and called. The function binding is done at run-time and depends on the class of the object to which the pointer points. This can be contrasted with compile-time binding where the compiler decides at compile-time which function to use. The canonical example is a base class `shape` and a virtual function `draw()`. The classes `square` and `triangle` are derived from `shape`. Suppose `x` is defined as `shape *x`;, at run-time `x` may point to a `shape`, a `square`, or a `triangle`. When `x->draw()` is executed the correct draw (`square` or `triangle`) will be bound and invoked.

Function and operator overloading permits the programmer to redefine, or overload, the meaning of both the standard binary and unary operators, as well as user defined functions. The compiler determines which function to use based on the number and type of the arguments.

Classes in C++ are defined in a manner similar to *structs* in C. The class int_stack

```
class int_stack {
protected:
   int max_elems, top;
   int *data;
public:
   int_stack(int size = 50);
   void push(int);
   int pop();
};
```

has three protected member variables defined, `max_elems`, `top`, and `data`. They may not be directly manipulated by users of instances of `int_stack`, but may be used by derived classes. The *constructor* for `int_stack`, `int_stack(int size)`, is called whenever a new instance is created. Constructors usually initialize private data structures and allocate space. Instances are created when a variable comes into scope, e.g., `{int_stack x(40);}`, or when instances are allocated on the heap, e.g., `int_stack *x = new int_stack(30);`. The member functions `push(int)` and `int pop()` operate on the stack and are the sole mechanism to manipulate private data.

To illustrate member function invocation, suppose that `x` is an instance of `int_stack`. Member functions are invoked using either the dot notation, `x.push(5);`, or if `x` is a pointer, the arrow notation, `x->push(5)`;.

[13] B. Stroustrup, ”What is Object-Oriented Programming?” *IEEE Software*, pp. 10-20, May, 1988.
[14] B. Stroustrup, *The C++ Programming Language*, 2nd ed. Addison-Wesley, Reading, Mass., 1991.