An Introduction to Parallel Object-Oriented Programming with Mentat

Andrew S. Grimshaw

Computer Science Report No. TR-91-07
April 4, 1991

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

An Introduction to Parallel Object-Oriented Programming with Mentat

Andrew S. Grimshaw

Abstract

Mentat is an object-oriented, parallel computation system designed to provide large amounts of easy to use parallelism for distributed systems. Mentat alleviates most of the burden of explicit parallelization that message passing systems typically place on the programmer. Further, Mentat programs block only if specific data dependencies require blocking, thereby greatly increasing the degree of parallelism attainable over that from RPC systems.

The Mentat Programming Language, an extension of **C++**, simplifies writing parallel programs by extending the encapsulation provided by objects to the encapsulation of parallelism. Users of Mentat objects are unaware of whether member functions are carried out sequentially or in parallel. In addition, member function invocation is asynchronous (non-blocking), the caller does not wait for the result. It is the responsibility of the compiler, in conjunction with the run-time system, to manage all aspects of communication and synchronization. The underlying assumption is that the programmer can make better granularity and partitioning decisions, while the compiler and run-time system can correctly manage communication and synchronization. By splitting the responsibility between the compiler and the programmer we exploit the strengths of each, and avoid their weaknesses.

Mentat has been implemented on three architectures that span the MIMD spectrum, a network of Sun workstations (loosely coupled), the Intel iPSC/2 (tightly coupled), and the BBN Butterfly (shared memory). Mentat programs are source compatible between supported architectures. Even on an 8 processor network of Sun workstations, speed-ups in excess of four, in comparison to optimized sequential **C** code, are consistently seen for Gaussian elimination using partial pivoting.

This paper describes the Mentat approach to parallelism, the Mentat Programming Language, an overview of the run-time system, and performance data on the above architectures. Examples that illustrate the major language features, and how the features support parallelism encapsulation are provided.

An Introduction to Parallel Object-Oriented Programming with Mentat

Andrew S. Grimshaw
Department of Computer Science
University of Virginia
Charlottesville, VA

1. Introduction

The last several years have witnessed the development of very high performance MIMD architectures. Examples include the Intel iPSC/2 [1], the BBN Butterfly [2], and recently the Intel i860 cube [3]. The peak performance of the most recent entry, a 128 i860 node hypercube, is in excess of 7.5 giga-flops, rivaling the performance of high-end supercomputers such as the Cray Y-MP (2.7 giga-flops) and the NEC SX-2 (1.3 giga-flops) [4]. In the near future a 2048 i860-node computer using a mesh interconnection network will be built. When complete it will have a peak performance of over 122 giga-flops [3]!

Concurrent with the development of the new high performance parallel supercomputers has been the rapid development in distributed systems technology, both hardware and software. In terms of hardware the now mature Ethernet LAN technology combined with low cost workstations has led to the proliferation of networks of workstations. Efforts have been made to combine these workstations to provide the users with the illusion that they are working with one large virtual machine that possesses the resources of all of its constituent hosts, e.g., memory, CPU cycles, and disk space. The goal of providing the illusion of a virtual machine has been best met with respect to file systems. One of the most successful examples is the NFS [5,6]. NFS provides the illusion of one large file space transparently accessible from any node in the system. The goal of providing transparent access to CPU resources in distributed systems is more elusive.

Common to both the high performance parallel architectures and the more loosely coupled distributed systems is the problem of writing software to exploit the available CPU resources. Writing software for parallel machines has proven to be far more difficult than writing sequential software. The difficulty is largely due to the low level of abstraction at which programmers must operate in order to utilize the resources, or, in the case of RPC systems, the inappropriateness of the abstraction, as typically implemented, for realizing parallelism. Currently many of these systems are programmed with traditional languages such as **C** and **FORTRAN** that are extended with library calls to support shared memory or message passing programming. This means that the programmer is responsible for managing problem decomposition, scheduling, communication, and synchronization. For many programmers this makes writing parallel programs too difficult. Proposals to solve this problem range from automatic program transformation systems that extract parallelism from sequential programs [7,8], to the use of side-effect free languages [9,10], to the use of languages and systems where the programmer must explicitly manage all aspects of communication, synchronization, and parallelism, albeit with high-level language support [11,12,13]. What is needed are language abstractions that permit the programmer to express algorithms using conventional techniques and which capture his domain knowledge of appropriate granules of computation, yet provide the compiler with enough information to automatically detect and manage medium-grain parallelism.

Mentat is a system currently running at the University of Virginia which was designed to solve the problem of writing parallel software for parallel and distributed systems. Mentat attacks the problem by providing high-level abstractions for parallelism that closely match the way programmers are accustomed to writing programs. Mentat provides an efficient, machine independent, portable set of run-time support facilities and language abstractions that provide a clear separation between the user and the physical system. Mentat supports location transparency, object-oriented design, parallelism encapsulation, spatial and temporal uncoupling, automatic detection and management of communication and synchronization, distributed continuations, and dynamic process creation with scheduling transparency.

The Mentat approach combines a medium-grain, data-driven computation model with the object-oriented programming paradigm and provides automatic detection and management of data dependencies. The graph based, data-driven computation model supports high degrees of parallelism and a simple decentralized control, while the use of the object-oriented paradigm permits the hiding of much of the parallel environment from the programmer. Because Mentat uses a data-driven computation model, it is particularly well-suited for message passing distributed systems

There are two primary components of Mentat: the Mentat Programming Language (MPL) [14,15] and the Mentat run-time system[16]. The MPL is an object-oriented programming language based on C++ [17] that masks the complexity 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. The programmer is responsible for identifying those object classes that are of sufficient computational complexity to allow efficient parallel execution. Instances of Mentat classes are used exactly like C++ classes, freeing the programmer to concentrate on the algorithm, not on managing the environment. 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. 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 is 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 granularity and partitioning, while the compiler can better manage synchronization. This simplifies the task of writing parallel programs, making parallel architectures more accessible to non-computer scientists.

As an example of the power of the Mentat approach, suppose we have an instance `matrix_op` of a `matrix_operator` class with the member function `mpy` that multiplies two matrices together and returns a matrix. Suppose `x, A,B,C,D` and `E` are `matrix` pointers, and that `B,C,D` and `E` point to matrices. Consider the sequence of statements

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

The two matrix multiplications `(B*C)` and `(D*E)` 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.

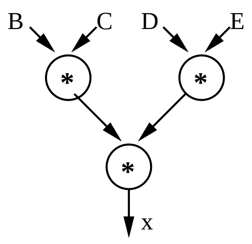


Figure 1. Parallel Execution of Matrix Multiply Operations.

In general, the intermediate result x could be sent to any number of destinations. In Mentat, member function invocations automatically proceed concurrently whenever data dependencies permit. Further, the implementation of the individual multiply operations could be parallel, increasing the level of parallelism. The key point is that the programmer need not concern himself with data dependence detection, the compiler does it!

The remainder of the paper is organized as follows. In Section 2 we introduce the Mentat Programming Language (MPL). Several examples are presented that show how the different language constructs are used, and more importantly, how much and what type of parallelism results from their use. Section 3 provides an overview of the Mentat run-time system and run-time system architecture. We pay particular attention to the layered nature of the run-time system, and to the portability of user applications that the layered approach supports. Section 4 presents performance data for two applications on both a distributed system consisting of Sun workstations, and on the Intel iPSC/2. Section 5 compares the Mentat approach to other approaches to the parallel software problem, particularly those targeted at loosely coupled distributed systems. Section 6 summarizes our work and presents our current and future efforts with respect to Mentat.

2. The Mentat Programming Language (MPL)

The object-oriented approach to programming has become very popular. The object-oriented approach promises to both simplify software development and to facilitate software reuse.[1] The object-oriented paradigm combines object-based techniques that hide (encapsulate) the implementation code and data structures in classes with the use of inheritance, specialization, and polymorphism [19-21]. The encapsulation of implementation code and data structures into classes makes software modular, and makes it possible to change the implementation without affecting users of the class.

The object-oriented paradigm is ideal for 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. Object member function invocations can be easily converted into remote procedure calls when necessary. A user of an object invokes the object member functions in the same way independently of whether the object invocation is local or remote, thus obtaining location transparency. The above properties have made distributed object-oriented systems based on RPC very popular. Examples include Argus [22], Clouds [23], Eden [24], ARTS [25], and many others [12]. The primary difference between Mentat and other distributed object-oriented systems is that Mentat is designed for parallelism and thus treats member function invocation differently.

The Mentat programming language is an extension of C++. We chose **C++** from among the object-oriented languages for several reasons. First, **C++** is a very efficient language; the implementation of the language features does not impose a performance penalty *vis a vis* **C** performance. This is important because the objective in parallel processing is performance.

[1] For a discussion of the merits of the object-oriented paradigm see [18-19]. See [20] for a discussion of the difference between *object-oriented* and *object-based* languages.

Second, the language does not already have language constructs for concurrency and parallelism as, for example, **ADA** does. This is important to avoid having two different parallelism constructs in our language. Third, **C++** supports typeless pointers, making it easier to write library routines that directly manipulate low level memory structures.

The remainder of this section is divided into seven parts. We begin with a short primer on **C++**, restricting the discussion to those features of **C++** that are needed to understand the MPL extensions. Next, we discuss the goals of the four language extensions. We then present Mentat class definitions, Mentat object instantiation, Mentat object member function invocation, the *return to future* construct, and the *select/accept* statement. Throughout this section we develop several examples that illustrate MPL concepts, and show how the parallelism is realized.

2.1. A Brief Introduction to C++

C++ is an object-oriented extension of C developed by Bjarne Stroustrup [17]. C++ supports classes with private data, public data, member functions that operate on instances of classes, multiple inheritance, polymorphism, and function and operator overloading. Classes in C++ are defined in a manner similar to *structs* in C. The class `int_stack`

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

has three private *member variables* defined, `max_elems`, `top`, and `data`. They may not be directly manipulated by users of instances of `int_stack`. The *constructor* for `int_stack`, `int_stack(intsize)`, 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);`. For a more complete description of C++ see [17].

2.2. Goals of the Language Extensions

The MPL was designed to meet four goals. First and foremost, the MPL would be object-oriented [18-21]. We would extend the usual notions of data and method encapsulation to include *parallelism encapsulation*.

Second, the language constructs would have a natural mapping to the macro data flow model [26,27][2]. The responsibility for performing the mapping would not be the programmer's. Instead, the compiler and run-time system would perform the mapping. The programmer would not have to make scheduling decisions, manage communication or synchronization, or manually construct and manage program graph execution.

Third, the concepts used as the basis of the extensions would be applicable to a broad class of languages so that the Mentat approach could be easily used in other contexts. Fourth, the syntax and semantics of the extensions would follow the pattern set by the base language, maintaining its basic structure and philosophy whenever possible.

[2]The macro data flow model is the computation model underlying Mentat. It is a medium grain, data driven computation 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. Data flows along the edges. When an actor has data on each input arc it may execute, consume the data, and produce new data on its outgoing edges. During the course of its computation an actor may elaborate its vertex into a subgraph of arbitrary complexity. The subgraph must have a sink that can be connected to the actors' outgoing edges in the original program graph.

These goals have been met in the MPL by extending the C++ language in three ways. The extensions are 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 system do the rest. This is accomplished using the keyword **MENTAT** 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 macro data flow graphs (data dependency graphs) in which the vertices are Mentat object member function invocations, and the arcs are the data dependencies found in the program. Thus we generate inter-object parallelism in a manner largely transparent to the programmer. Communication and synchronization are managed by the compiler.

2.3. Mentat Class Definition

There are three issues to be resolved with respect to Mentat classes, 1) how to determine which classes should be Mentat classes, 2) how to syntactically specify that a class is a Mentat class, and 3) how to determine and specify what type of Mentat class should be used.

When should a class be a Mentat class? There are three types of classes that should be Mentat classes, those whose member functions are computationally expensive, those whose member functions exhibit high latency (e.g., IO), and those that hold state information that needs to be shared by many other objects (e.g., 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, because 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 [28] synchronization, providing synchronized access to their state.

Syntactically, the specification of a Mentat class is simple. We add the keyword `MENTAT` to the class definition. The class may be further specified to be `PERSISTENT` or `REGULAR`. Instances of `PERSISTENT MENTAT` classes persist from invocation to invocation, i.e., they maintain state information. `REGULAR MENTAT` class instances do not maintain state information; their member functions are pure functions. Thus the system is free to instantiate new instances of `REGULAR` classes at will. `REGULAR` classes may have local variables much as procedures do. By default a Mentat class is a `REGULAR` class.

Choosing the appropriate type of Mentat class is easy. If the class is stateless, it should be a `REGULAR` class. If the class requires maintaining state information, or if use of state information enables a more efficient implementation, the class should be a `PERSISTENT` class. A `PERSISTENT` class may be further specified to be a `SEQUENTIAL PERSISTENT` class. Instances of `SEQUENTIAL PERSISTENT` classes differ from instances of `PERSISTENT` classes in that invocations of `SEQUENTIAL PERSISTENT` instances are guaranteed to be executed in the order encountered in the program *even when there are no data dependencies to indicate a particular execution order to the compiler*.

Example 1

Below, a PERSISTENT Mentat class shared_queue is defined, with member functions enq(int) and int deq(). Note that, except for the keywords PERSISTENT MENTAT, this is a legitimate **C++** class definition.

```
PERSISTENT MENTAT class shared_queue {
  // Private data structures to implement the queue.
  int head, tail;
  int *data;
public:
  void  enq(int value);
  int deq();
}
```

2.4. Mentat Object Instantiation

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 let us examine instantiation in **C++**. Consider the **C++** fragment:

```
{ // A new scope
  shared_queue the_queue;
} // end of scope
```

In **C++**, when the scope in which the_queue is declared is entered, a new instance of shared_queue is instantiated. In the MPL, if shared_queue is a Mentat class, then the_queue is a *name* of a Mentat object of type shared_queue, it is not the instance itself.

Names (e.g., the_queue) 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 a Mentat class 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.

There are three ways a Mentat variable (e.g., `the_queue`) 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 operation. Both the `create()` member function and the `bind()` member function are common to all Mentat classes. They are inherited from a base class that is logically a superclass for Mentat classes. Examples of each mechanism are shown in Figure 2. Assume the definition `shared_queue the_queue;`.

```
(a)  the_queue.create();
(b)  the_queue.create(COLOCATE another_object);
(c)  the_queue.create(DISJOINT object1, object2);
(d)  the_queue.create(HIGH_COMPUTATION_RATIO);
(e)  the_queue.bind(THIS_HOST);
(f)  the_queue = some_function();
```

Figure 2. Binding Mentat Objects.

The `create()` call tells the system to instantiate a new instance of the appropriate class. There are four flavors of `create()`. When `create()` is used as in Figure 2 (a) the system will choose on which processor to instantiate the object [29]. 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 2 (b), the programmer has specified that the new Mentat object should be placed on the same processor as the object `another_object`. In Figure 2 (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 2 (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.

The `bind(scope)` call can be used when the programmer wishes to bind a name to an already existing instance of the class rather than creating a new instance. This is useful, for example, when trying to locate and bind to a persistent server. The scope parameter is used to specify the scope of the system's search for an instance. The search may be system-wide, or be confined to this host, or to this sub-network/cluster. If the system cannot find an existing instance then the variable is left unbound.

The `destroy()` member function destroys the Mentat object and unbinds the Mentat name. It does not invalidate names that point to the object.

2.5. Mentat Object Member Function Invocation

In this section we introduce the semantics of Mentat object member function invocation and illustrate how Mentat supports the automatic management of inter-object parallelism. Inter-object parallelism is the concurrent execution of member functions whenever data dependencies permit. We say that we provide inter-object parallelism encapsulation because the parallelism is transparent to the user. We begin with an example illustrating a simple RPC, and then proceed to illustrate inter-object parallelism using a pipeline example.

The invocation syntax for Mentat objects is the same as in **C++**. The semantics are slightly different because Mentat objects have disjoint address spaces. Mentat class member functions always use call-by-value semantics. This is true even when pointers or references are passed. When pointers are used, the object (or structure) pointed to is sent to the callee. If the object pointed to 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.

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();
  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 a simple RPC to a Mentat object member function. As such it is not particularly novel, or parallel. The difference between Mentat and a traditional RPC is what happens when an RPC call is encountered. In traditional RPC the arguments are marshalled, 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. Note, though, that if `x` is never used locally (except as an argument to a Mentat object member function invocation) then the caller never blocks and waits for `x`. Indeed `x` might never be sent to the caller, `x` might only be sent to the Mentat object member functions for which `x` is a parameter. It is important to note that these decisions, as well as all communication and synchronization, are handled completely by the compiler and run-time system. The programmer is left free to concentrate on the application, not on the details.

Example 3

This example illustrates the power of the Mentat approach by constructing a simple pipeline process. For this example we define the Mentat class

```
REGULAR MENTAT class data_processor
public:
  data_block* filter_one(data_block*);
  data_block* filter_two(data_block*);
}
```

The member functions `filter_one()` and `filter_two()` are filters that process blocks of data. The exact function is not important. Consider the code fragment in Figure 5.

```
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((i*BLK_SIZE),BLK_SIZE);
  res = dp.filter_one(res);
  res = dp.filter_two(res);
  out_file.write((i*BLK_SIZE),BLK_SIZE,res);
}
```

Figure 5. A Pipelined Data Processor.

This 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".

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 four times the member function execution time plus seven times the communication time. It is 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. We arrived at the figure of 10 time units by first observing 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. 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). 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 faster than a pure RPC (75 time units) because we don't send intermediate results to the caller. However, as `MAX_BLOCKS` increases the average time per iteration drops, and approaches 10 time units.

There are four items to note from this example. First, the variable `dp` is a REGULAR MENTAT class. Thus the system is free to instantiate new instances at will. At any given time in this example there will be two instances executing. Second, the main loop may have executed to completion (all `MAX_BLOCKS` iterations) before the first write has completed! Third, 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. Fourth, 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.

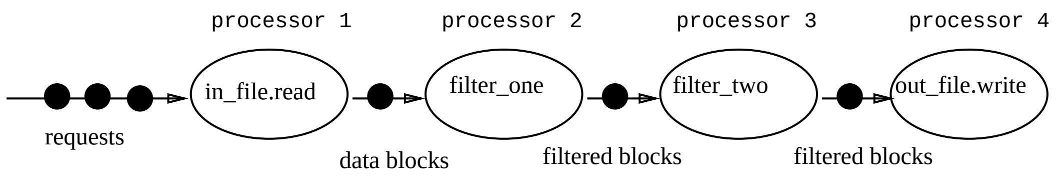


Figure 6. Four Stage Filter Pipeline.

Now consider the effect of quadrupling the time to execute the filters from 10 to 40 time units. The time to execute the traditional RPC version goes from 75 to 135 time units. But 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. Thus, in the above scenario, there would be eight instances of the `data_processor` class active at a time, four performing filter one, and four performing filter two.

Before we continue, a few observations are in order. First, the detection and management of data dependencies is fully automatic. This frees the programmer to concentrate on the semantics of the program. Second, the scheduling of regular object invocations is fully automatic. Manual scheduling can be achieved by making the `data_processor` class a persistent class and instantiating instances on the desired processors. Third, in addition to the automatic detection of inter-object parallelism opportunities, each of the invoked member functions may be internally parallel, providing even more parallelism. The hiding of such parallelism is called intra-object parallelism encapsulation.

2.6. Return to Future - rtf()

The *return to future* (`rtf()`) function is the Mentat analog to the **C** `return`. Mentat member functions use the `rtf()` as the mechanism for returning values. The value returned is forwarded to all member functions that are data dependent on the result, and to the caller *if necessary*. In *Example 3* when `filter_one()` performs its `rtf()` the result is sent on to `filter_two()`. A copy of the result is *not* sent back to the caller. In general copies may be sent to several recipients.

While there are many similarities between `return` and `rtf()`, `rtf()` differs from a `return` in three significant ways. First, a `return` returns data to the caller. `Rtf()` may or may not return data to the caller depending on the data dependencies of the program. If the caller does not use the result locally, then the caller does not receive a copy. 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. By making the result available as soon as possible we permit data dependent computations to proceed concurrently with the local computation that follows the `rtf()`. Third, in **C**, before a function can `return` a value, the value must be available. This is *not* the case with an `rtf()`. Recall that when a Mentat object member function is invoked, the caller does not block, rather we ensure that the results are forwarded wherever they are needed. Thus, a member function may `rtf()` a "value" that is the result of another Mentat object member function that has not yet been completed, or perhaps even begun execution. Indeed, the result may be computed by a parallel subgraph obtained by detecting inter-object parallelism.

The next two examples illustrate the use of `rtf()` and how intra-object parallelism is realized.

Example 4

Consider a transaction manager (TM) that receives requests for reads and writes, and checks to see if the operation is permitted. If it is permitted, the TM performs the operation via the data manager (DM) and returns the result. Below we illustrate how the read operation might be implemented.

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

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

Example 5

This example illustrates intra-object parallelism encapsulation. Consider a matrix class that performs matrix operations.

```
REGULAR MENTAT class matrix_operator {
public:
  loc_matrix* mpy(loc_matrix* mat1, loc_matrix* mat2);
  loc_matrix* combine_results(int pieces, loc_matrix* mat1, ...);
};
loc_matrix* matrix_operator::mpy(loc_matrix* mat1, loc_matrix* mat2){
int dim, i,no_rows=0;
matrix_operator worker;
loc_matrix *res[MAX_PIECES], *pm, *answer;
// Determine the # pieces using a heuristic
for (i=0;i<pieces;i++) {
  rows=dim/pieces;
  if (dim%pieces>i) rows++;
  pm = new loc_matrix(mat1->get_r_ptr(no_rows), dim, rows);
  res[i] = worker.mpy(pm[i], mat2);
  delete pm;
  no_rows += rows;
};
switch (pieces) {
  case 2:rtf(worker.combine_results(2,res[0],res[1]);
  break;
  case 4:rtf(worker.combine_results(4,res[0],res[1],res[2],res[3]);
  break;
  ...
}; }
```

Suppose that the matrix multiply operation is invoked as in the code fragment below.

```
loc_matrix* a,b,c;
// Assume a,b are set up with values
matrix_operator A;
c = A.mpy(a,b);
c->print();
```

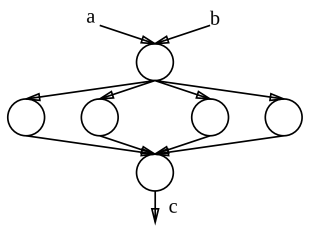
The `A.mpy(a,b)` results in a multiply operation being invoked. Suppose that the hueristics selected four pieces. Then the `A.mpy` operation would be expanded into the parallel subgraph shown in Figure 7 (a). The parallelism is encapsulated inside of the member function invocation. The user is unaware of the expansion.

Example 6

This example illustrates both inter and intra-object parallelism. Consider the fragment:

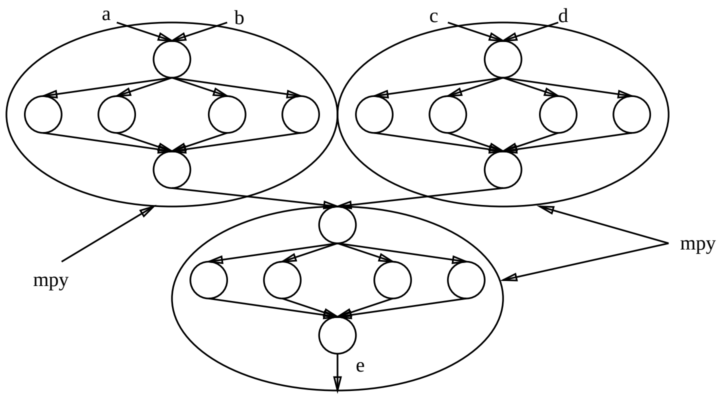
```
loc_matrix* a,b,c,d,e;
// Assume a,b,c,d are set up with values
matrix_operator A;
e = A.mpy(A.mpy(a,b),A.mpy(c,d));
```

Intra-object encapsulation is realized through the parallel execution of each of the multiplies as before. Inter-object parallelism is detected by observing that the two inner-most multiplications may be executed in parallel. These are combined in the execution graph of Figure 7 (b). Once again note that the user of the `matrix_operator` class is not responsible for managing any aspect of the parallel execution, nor need he/she even be aware that it is parallel!



Intra-Object Parallelism, Matrix Multiply Expansion

(a)



Intra and Inter Object Parallelism

(b)

Figure 7. Parallel Execution Graphs for Matrix Multiply.

2.7. Select/Accept

The *select/accept* construct in the MPL is similar in many ways to the **ADA** [30] *select/accept*. MPL *select/accept* statements are used to selectively choose which member functions are candidates for execution, i.e., those member functions for which the object will *accept* an invocation. Unlike the **ADA** *select/accept* the MPL *select/accept* is symmetric: the caller can specify the modules with which it will interact, and can specify conditions that must be true of the caller's arguments.

As in **ADA**, each accept statement may have a *guard* clause that must evaluate to **true** in order for the guarded member function to be a candidate for execution. In **ADA**, guards are limited to expressions based on the local state of the object. Thus an object may not examine the parameters of a member function invocation as part of guard evaluation. This often leads to awkward constructs to examine the arguments *after* the call has been accepted, and somehow postpone servicing the request until a later time.

MPL guards may contain an arbitrary side-effect free expression based on local state *and* the actual parameters. The guard may also contain the *name* of the caller. As an example, consider a bank account object and a member function `withdrawal(int account,int amount)`. We might wish to restrict withdrawals to less than the current balance of the account, .e.g.,

```
select {
  (amount<balance(account)):
    accept withdrawal(int account,int amount);
  break;
  :accept deposit(int account,int amount);
  break;
};
```

The guard for the `withdrawal` has the effect of deferring overdrafts until sufficient funds have been deposited to the account via `deposit`. The ability to selectively receive invocations was inspired by PLITS [31].

Guards are evaluated in the order of their *priority*, from high priority to low priority. Priorities are in the range (-`MAXINT`) to `MAXINT`. The default priority is zero. Guards within a given priority level are evaluated in a non-deterministic order until a guard evaluates to true. If none of the guards evaluates to **true** then the object blocks until a guard evaluates to true. The priorities can be used to force an order of evaluation on the guards. Using the example above, if we wanted to prioritize deposits over withdrawals we could change the code as shown below.

```
select {
  [0] (amount<balance(account)):
    accept withdrawal(int account,int amount);
  break;
  [10] :accept deposit(int account,int amount);
  break;
};
```

Select/accept statements can also be used to test, without blocking, whether there are pending member function invocations. This is accomplished by using *test* instead of *accept*. *Tests* are non-blocking. Thus if there are no pending requests that satisfy a *select/test* statement then the object does not block.

2.8. Restrictions

There are several restrictions on Mentat class definition and member variables. These restrictions derive from the fact that instances of Mentat classes are independent objects. There is no shared memory. The independence of address space between Mentat objects necessitates the imposition of four restrictions on Mentat classes and their use.

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

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

Third, programmers cannot assume that pointers to instances of Mentat classes point to the member data for the instance. This is a common trick used by **C** programmers, but it will not work in Mentat.

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

3. The Run-Time System

The Mentat run-time system provides run-time support to Mentat applications[16]. The run-time system can be divided into two distinct sets of services. First are the library routines and data structures that support data dependence detection, guard evaluation, select/accept management, and communications services. The second set consists of Mentat objects that provide scheduling, Mentat object instantiation, and token matching services[3].

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 process, memory, **C** library, and interprocess communication (IPC) services. To date, Mentat has been hosted on 4.3 BSD Unix [32], NX/2 [1] (the operating system on the Intel iPSC/2), and will soon be complete on Mach [33] and Mach-1000 [2] (BBN's dialect of Mach).

The logical structure of a Mentat system is that of a collection of hosts communicating through an interconnection network (see Figure 8). Each host is able to communicate with any other host via the interconnection network, although not necessarily at uniform cost[4].

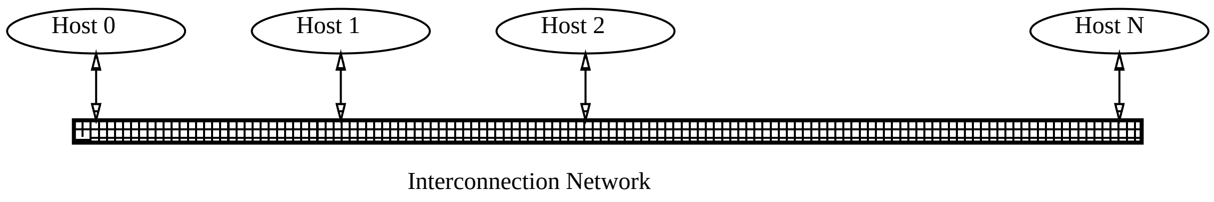


Figure 8. Mentat Run-Time System Logical Structure.

The logical interconnection network is provided by the lowest layer of the run-time system, **MMPS**. MMPS (Modular Message Passing System [34]) provides an extensible point-to-point message service that reliably delivers messages of arbitrary size from one process to another[5]. MMPS provides a uniform message passing interface that we have ported to physical interconnection networks that span the performance spectrum, from high latency/low bandwidth/high contention (Ethernet using IP datagrams), to low latency/low bandwidth/low contention (the iPSC/2), to a very low latency/very high bandwidth/low contention network (the

[3] Token matching is the gathering together the arguments of an invocation of a REGULAR object member function onto one host.
[4] The scheduler [29] uses communication cost information in making scheduling decisions.
[5] Arbitrary in that insufficient memory at the receiver is the only limitation. A process is as defined by the host operating system.

BBN Butterfly using shared memory). MMPS also supports communication between peers on different operating systems, e.g., Mach, Mach-1000, and Unix, although the CPU architectures must be the same. This inter-operating system capability permits us to construct Mentat systems composed of different types of hosts (Suns, BBN) running different operating systems.

Each host has a complete copy of the run-time system server objects. These include the *instantiation manager* 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, and is discussed in [29].

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 copy the sources *unchanged* to the Intel iPSC/2, recompile them, and execute them. 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.

4. Performance

4.1. Primitive Operations

The performance of Mentat, and of the Mentat approach, hinges on the speed with which primitive operations can be performed. In particular, message operations and the time to detect data dependence and construct program graphs must all incur little cost, or overhead will swamp the gains from parallelism. There are two primary sources of overhead, communication, and the Mentat overhead associated with program graph construction, argument marshaling, and select/accept execution. In Tables T-1 and T-2 below the time spent in each of these activities is shown for a Sun 3/60 and a node on the Intel iPSC/2 respectively. The message transport cost includes the host operating system scheduling and task switch overhead. For information on the communication costs see [34].

Function	Time
Transport Message (one way)	5.7mS
Null RPC	14.0mS
Mentat overhead	2.6mS

Table T-1. Component Timing Results (Sun 3/60).

Function	Time
Transport Message (Send)	0.8mS
Null RPC	2.8mS
Mentat overhead	1.2mS

Table T-2. Component Timing Results (iPSC/2).

We compute the Mentat overhead by subtracting from the Null RPC time two times the message transport cost (request & reply). The largest cost operation is communication.

4.2. Applications

Nice computation and programming models aside, the bottom line for parallel and distributed systems is performance. As of this writing we have implemented the Mentat run-time system and run benchmarks on a network of Sun workstations and on a sixteen node Intel iPSC/2 Hypercube. Speed-ups for two benchmarks on each of the supported architectures are given below. In each case the speed-up 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. The two benchmarks are matrix multiply and Gaussian elimination. Each benchmark was executed for several matrix dimensions., e.g., 100×100, 200×200. Matrix multiply and Gaussian elimination are used because they are *de facto* parallel processing benchmarks.

4.3. Execution Environment

The data were collected in two different environments: a network of Sun workstations and a Hypercube. 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.

The Intel iPSC/2 is configured with sixteen nodes. Each node has one megabyte of physical memory and an 80387 math co-processor. The nodes were **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 speed-ups for matrix multiply are shown in Figures 7 and 8 below. The algorithm (and application source) is the same for both systems. Suppose the matrices *A* and *B* are to be multiplied. Suppose *A* is the larger of the two. *A* is divided into *n* pieces $A_i...A_n$, where *n* is the number of processes to use. A copy of *B* and one of the A_i's are used as parameters to Mentat object invocations. The results of the invocations are merged together and sent to computations that are dependent on the result of the *A*B* operation.

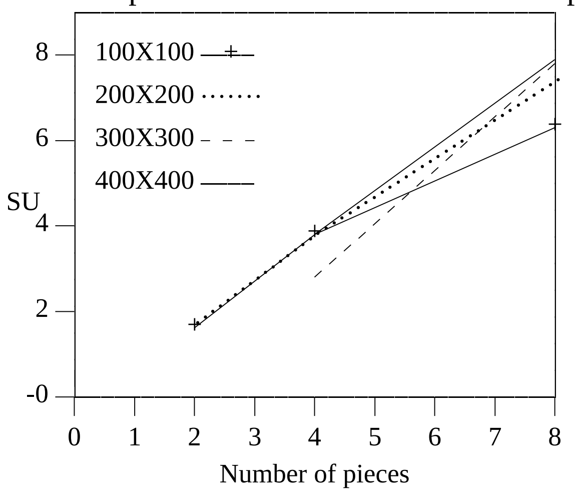


Figure 9. Sun network matrix multiply

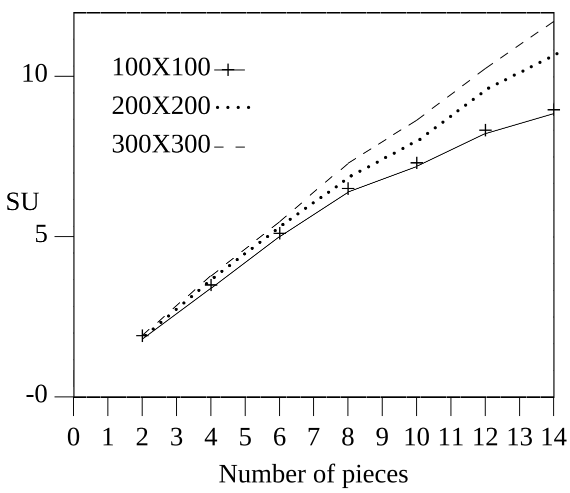


Figure 10. Hypercube matrix multiply

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 *sblocks* 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 requires frequent communication and synchronization. The effect of frequent synchronization can be clearly seen when the speed-up results for Gaussian elimination in Figures 9 and 10 are compared to the results for matrix multiply.

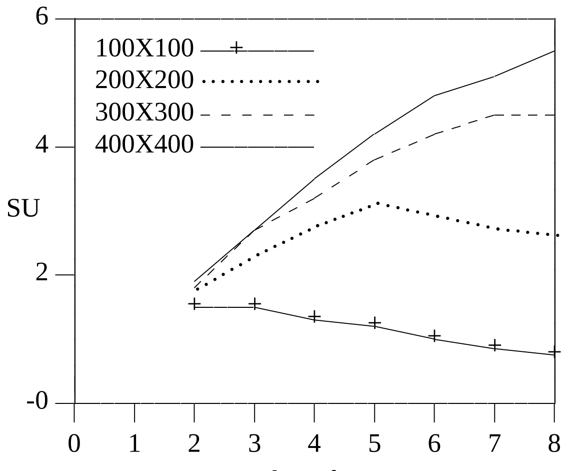


Figure 11. Sun Gaussian

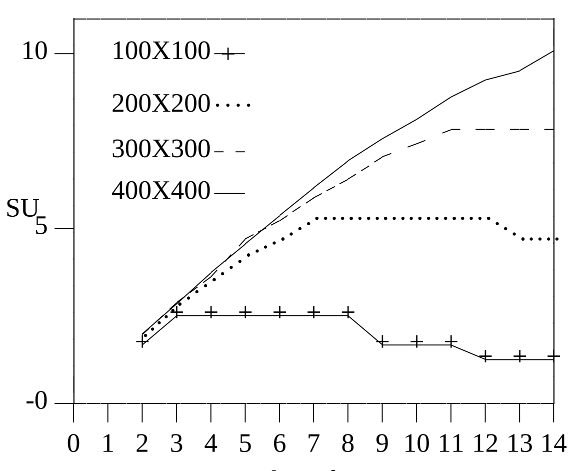


Figure 12. Hypercube Gaussian

5. Related Work

Mentat is built on top of a message passing system. Therefore, message passing systems are at least as powerful as Mentat. In order to compare a pure message passing system there are two questions that must be answered. First, can Mentat emulate message passing systems? In other words, are there mechanisms in Mentat that can be used as *send* and *receive*? Second, is Mentat easier to program than message passing systems that provide only *send* and *receive*?

The answer to the first question is a definite yes. *Send* and *receive* can be easily simulated by providing appropriate member functions for Mentat classes and by using an appropriate *select/accept* statement.

The answer to the second question is more subjective. If you don't think that programming using *send*/*receive* requires any special effort, then the mechanisms used by Mentat to hide the underlying message passing scheme are of limited utility. We feel that Mentat provides an easier-to-use interface by managing much of the complexity inherent in message passing systems. Programming in Mentat vs. programming using just *send*/*receive* can be compared to the difference between programming in a high-level language vs. programming in assembly language. While it is true one can do anything in assembly one can do in a high-level language, it is usually easier to code and debug high-level languages.

5.1. Object-Oriented Distributed Systems

Mentat differs from the majority of Object-Oriented distributed systems, such as [22-25,35-38], in the way remote member function invocations are managed. Much of the difference can be accounted for by the difference in objectives between Mentat and the others. Mentat strives for parallelism as opposed to distribution [24,38], fault-tolerance [22,23,36], or real-time operation [25]. Most object-oriented distributed systems provide a traditional, call, block, continue, RPC [39] semantics. Mentat supports both more parallelism and potentially less message traffic than the traditional approach by exploiting intra-object and inter-object parallelism opportunities and by sending intermediate results only where needed. For a good survey of programming languages for distributed systems see [12].

5.2. Promises

Promises [40] offers an extension to traditional RPC that allows multiple invocations of a remote procedure to be pipelined, not delaying execution of the caller until the result is actually needed. As shown in *Example 3* Mentat supports pipelining as a subset of its capabilities. In some respects Promises is similar to the Mentat approach. One principle difference is that the Promises approach does not allow generalized graph structures to be built. The results of invocations must first return to the caller before they can be used in subsequent invocations. More importantly, programmers must explicitly manage the results of the pipelined executions differently from other variables. The programmer must check if the result has arrived and an actual value is available, and block if it has not. Mentat does not require the programmer to manage synchronization by checking if the result is available. Instead, the programmer uses the variable and the compiler generates code to perform the checking.

5.3. Implicit vs. Explicit Parallelism

Parallelism can be either automatically extracted from the sources by the compiler (implicit parallelism), or specified and managed by the programmer (explicit parallelism). The relative merit of the implicit and explicit schemes depends on whether the programmer or the compiler can better detect parallelism, and make decisions regarding granularity, synchronization, and communication.

In implicit schemes a conventional or functional language can be used by the programmer. Implicit schemes [41,42] offer the advantage that they often permit the automatic parallelization of "dusty decks". This is the approach used in Cedar [8,43], and in vectorizing compilers such as those for the Cray. Implicit schemes are usually better at detecting fine-grain parallelism, and not as successful at detecting large-grain parallelism.

Explicit parallelization schemes allow the programmer to control parallelism by providing parallel programming constructs to specify which portions of the program will be executed in parallel. Synchronization and communication are often the responsibility of the programmer in explicit schemes. The extra control and flexibility of explicit schemes often comes at the cost of extra complexity.

Mentat falls between the two extremes of fully automatic compilers and the assembly language-like fully explicit schemes. In Mentat, the granules of computation are explicitly defined. Programmers explicitly specify those object classes that possess an independent thread of control. However, the dependencies and parallel structure of the computation are implicit. Invocation, communication, and synchronization are handled automatically by the compiler and the run-time system. The compiler generates code to automatically construct macro data-flow graphs at run-time. Furthermore, Mentat does not preclude the use of special purpose or vectorizing compilers to compile the individual objects to run on special purpose hardware or vector machines. We feel that hybrid approaches, such as the approach used by Mentat, offer a good compromise between the programming simplicity of implicit schemes and the control and power of explicit schemes.

5.4. Generative Communication - Linda

Generative communication and Linda [44] have been proposed as a means of writing parallel and distributed software. Mentat differs from Linda two respects. First, the underlying model of computation is different. Linda uses the generative communication model in which processes communicate with one another by reading and writing to the shared *tuple space* (TS). The tuple space can be thought of as a shared associative memory. Mentat uses a data-driven, message passing, model of computation in which processes communicate by sending one another messages. Second, Linda explicitly exposes the programmer to problem decomposition, granularity, communication, and synchronization. Mentat hides communication and synchronization in member function calls, providing instead a familiar programming abstraction. Both systems provide location transparency, spatial and temporal uncoupling, and a form of continuation passing.

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. 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 and distributed architectures will be widely used until software tools are available that hide the complexity of the parallel environment from the programmer.

In this paper we have presented the Mentat approach to solving the parallel software problem. The key idea is to have the programmer use his domain knowledge to decompose the problem, and to use compiler technology to correctly manage scheduling, communication, and synchronization. By exploiting the strengths of both the programmer (domain knowledge) and the compiler (correctness) we can leverage the programmer's efforts, freeing him/her from detail, and increasing his/her productivity.

Our approach includes extending the object-oriented paradigm's encapsulation techniques to include parallelism encapsulation. The programmer uses his 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 transparently 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 with shared memory) 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 vs. assembly languages and for MPL vs. raw send and receive is whether the simplicity and ease of use are worth the performance penalty. We believe that they are.

Acknowledgements:

A project of this scope is not possible without the efforts of many people. I would like to thank all of the members of the Mentat group, Gorrell Cheek (validation suite and benchmarking), Doug Coppage (Mach port), Ed Loyot (MPL compiler), Dave Mack (MMPS), Jeff Prem (Mentat parallel I/O package), and Virgilio Vivas (*i_m* and scheduler). Special thanks to Jane W.S. Liu (U. of Illinois) for her comments and assistance. I would also like to thank the Institute for Parallel Computation at the University of Virginia for the use of their Intel Hypercube. Finally, thanks to our sponsors, the Information Procssing Technical Branch at NASA LARC.

References

[1] Intel Corporation, ‘‘iPSC/2 USER’S GUIDE’’, Intel Scientific Computers, Beaverton, OR, March 1988.

[2] BBN Advanced Computers Inc., ‘‘Mach-1000 Reference Manual’’, Cambridge, Mass., 1988.

[3] S. Squires, Keynote address, *Fifth Distributed Memory Computing Conference*, Charleston, SC., April 9-12, 1990.

[4] J. J. Hack,"Peak vs. Sustained Performance in Highly Concurrent Vector Machines", *IEEE Computer*, September, 1986.

[5] S. J. Mullender, *Distributed Systems*, ACM Press, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[6] E. Levy, and A. Silbershatz, "Distributed File Systems: Concepts and Examples", Computer Science TR-89-04, University of Texas at Austin, March 1989.

[7] K. Kennedy, "Optimizations of Vector Operations in an Extended Fortran Compiler", *IBM Research Report*, RC-7784, 1979.

[8] D. Kuck, R. Kuhn, B. Leasure, D. Padua, and M. Wolfe, ‘‘Dependence Graphs and Compiler Optimizations,’’ *ACM Proc. 8th Annual ACM Symposium on Principles of Programming Languages*, pp. 207-218, Jan., 1981.

[9] W. B. Ackerman, ‘‘Data Flow Languages,’’ *IEEE Computer*, vol. 15, no. 2, pp. 15-25, February, 1982.

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

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

[12] H. E. Bal, J. G. Steiner, and A. S. Tannenbaum,‘‘Programming Languages for Distributed Computing Systems,’’ *ACM Computing Surveys*, pp. 261-322, vol. 21, no. 3, September, 1989.

[13] C. M. Pancake, and D. Bergmark, "Do Parallel Languages Respond to the Needs of Scientific Programmers?", *IEEE Computer*, pp. 13-23, December, 1990.

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

[15] A. S. Grimshaw and E. Loyot,‘‘The Mentat Programming Language: Users Manual and Tutorial,’’ Computer Science TR-90-08, University of Virginia, April, 1990.

[16] A. S. Grimshaw,‘‘The Mentat Run-Time System: Support for Medium Grain Parallel Computation,’’ *Proc. of the 5th Distributed Memory Computing Conference*, pp. 1064-1073, Charleston, SC., April 9-12, 1990.

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

[18] J. D. McGregor, and T. Torson, guest eds., Special Issue on Object-Oriented Design, *CACM*, September, 1990.

[19] B. Stroustrup,"What is Object-Oriented Programming?," *IEEE Software*, pp. 10-20, May, 1988.

[20] P. Wegner,"Dimensions of Object-Based Language Design," *Proceedings of the 1987 Object-Oriented Programming Systems, Languages and Applications Conference*, ACM, pp. 168-182, October, 1987.

[21] A. Goldberg, and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[22] B. Liskov, and R. Scheifler, ‘‘Guardians and Actions: Linguistic Support for Robust, Distributed Programs,’’ *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, pp. 381-414, 1983.

[23] R. H. LeBlanc, and C. T. Wilkes, ‘‘Systems Programming with Objects and Actions,’’ *Proceedings 5th Distributed Computer Systems*, IEEE, pp. 132-139, 1985.

[24] A. Black, ‘‘Supporting Distributed Applications: Experience with Eden,’’ *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 181-193, December, 1985.

[25] C. Mercer, and H. Tokuda,‘‘The ARTS Object Model,’’ *Proceedings of the 11th Real-Time Systems Symposium*, Orlando, FL, Dec.4-7, 1990.

[26] J.W.S. Liu and A. S. Grimshaw, ‘‘An object-oriented macro data flow architecture,’’ *Proceedings of the 1986 National Communications Forum*, September, 1986.

[27] J.W.S. Liu and A. S. Grimshaw, ‘‘A Distributed System Architecture Based on Macro Data Flow Model,’’ *Proceedings Workshop on Future Directions in Architecture and Software*, South Carolina, May 7-9, 1986.

[28] C.A.R. Hoare,‘‘Monitors: An Operating System Structuring Concept,’’ *Communications of the ACM* pp.549-557, vol. 17, no. 10, October, 1974

[29] A. S. Grimshaw, and V. E. Vivas, ‘‘FALCON: A Distributed Scheduler for MIMD Architectures’’, *submitted to Symposium on Experiences with Distributed and Multiprocessor Systems*, Atlanta, GA, March, 1991.

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

[31] J. A. Feldman, ‘‘High Level Programming for Distributed Computing,’’*Communications of the ACM,* pp. 353-368, vol. 22, no. 6, January, 1979.

[32] W. Joy, E. Cooper, R. Fabry, S. Leffer, K. McKusick, and D.Mosher, ‘‘4.2BSD System Manual,’’ Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, July 1983.

[33] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, ‘‘Mach: A New Kernel Foundation for Unix Development’’, *Summer Usenix Conference Proceedings*, pp. 93-112, 1986.

[34] A. S. Grimshaw, D. Mack, and T. Strayer,"MMPS: Portable Message Passing Support for Parallel Computing," *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 784-789, Charleston, SC., April 9-12, 1990.

[35] M. B. Jones, and R. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Based Distributed Systems,’’ Carnegie Mellon University, 1986.

[36] D. L. Detlefs, M. P. Herlihy, and J. M. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++," *IEEE Computer*, December, 1988.

[37] A. S. Tanenbaum, and R. van Renesse, ‘‘Distributed Operating Systems," *ACM Computing Surveys*, pp. 419-470, vol. 17, no. 4, December, 1985.

[38] R. van Renessee, H. van Staveren, and A. Tannenbaum, "Performance of the World’s Fastest Distributed Operating System," *ACM OSR*, vol. 22, no. 4 (Oct.), 1984.

[39] B. J. Nelson,‘‘Remote Procedure Call,’’ Xerox Corporation Technical Report CSL-81-9, May, 1981.

[40] B. Liskov, and L. Shrira,"Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, pp. 260-267, June, 1988.

[41] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

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

[43] D. Kuck, D. Lawrie, R. Cytron, A. Sameh, and D. Gajski, ‘‘THE ARCHITECTURE AND PROGRAMMING OF THE CEDAR SYSTEM,’’ Cedar Document no. 21, University of Illinois at Urbana-Champaign, Department of Computer Science, August, 1983.

[44] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer*, pp. 26-34, August, 1986.