To appear in Procedings of the 5th Distributed Memory Computing Conference, Charleston, S.C. April 1990.

The Mentat Run-Time System: Support for Medium Grain Parallel Computation

Andrew Grimshaw

Computer Science Report No. TR-90-09
April 1990

This work was partially supported by JPL contract #957721 and by DOE under grant DE-FG05-88ER25063.

The Mentat Run-Time System: Support for Medium Grain Parallel Computation†

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

Abstract

The goal of Mentat is to provide easy-to-use parallelism to non computer-scientists. This paper presents the Mentat run-time system. The run-time system supports medium-grain, object-oriented computation using a data-driven computation model. Performance figures are included.

1. Introduction

As we move toward the existence of tera-op machines the need to cost-effectively build parallel application software grows more pressing. Although research tools have been built to assist in detecting or expressing parallelism at a fine-grain, we need techniques and tools to detect and express parallelism at a medium-grain size, a size that better matches the substantial processing power of the individual nodes in a MIMD style tera-op machine. Without such tools the result will be manually-created parallelism that is typically large-grain, or very fine-grain parallelism produced by automatic compilers.

One approach to the parallel software problem is **Mentat**[1-2]. Mentat combines a medium-grain, data-driven computation model with the object-oriented programming paradigm and provides automatic detection and management of data dependencies. The 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, non-shared memory architectures.

There are two primary aspects of Mentat: the Mentat Programming Language (MPL) [2] and the Mentat run-time system. The MPL is an object-oriented programming language based on C++ [3] that masks the difficulty of the parallel environment from the programmer. The granule of computation is the Mentat class instance, which consists of contained objects (local and member variables), their procedures, and a thread of control. 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. The data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the preprocessor 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 scientist researchers. See [2] for a more complete description of the Mentat programming language design and implementation.

This paper is concerned with the run-time system support required to execute Mentat programs. The run-time system supports parallel object-oriented computing on top of a data-driven, message-passing model. It supports more than just method invocation by remote procedure call (RPC). Instead the run-time system supports a graph-based, data-driven computation model in which the invoker of an object member function need not wait for the result of the computation, or for that matter, ever receive a copy of the result. The run-time system constructs program graphs, and allows selective message reception. Furthermore, the run-time system is portable across a wide variety of MIMD architectures and runs on top of the existing host operating system. The underlying operating system must provide process support and some form of inter-process communication. The run-time system is currently running on two systems: the Intel Hypercube using **NX/2** [4], and a network of Sun workstations using **UDP** packets and **BSD** Unix sockets [5-6].

The remainder of the paper is organized as follows. First, a brief overview of the run-time system architecture is presented. Next, the macro data flow model of computation is described. The macro data flow model is the model of computation underlying Mentat. Then, the run-time libraries are presented. The salient features of the MPL, as well as an example used throughout the rest of the paper, are presented to motivate the library features. The run-time system itself is then described in detail, followed by performance figures for the run-time system and two applications. The performance figures are presented for both the Intel Hypercube and the Sun workstation implementations. We conclude the paper with some observations and plans for future work.

†This work was partially supported by JPL contract #957721 and by DOE under grant DE-FG05-88ER25063.

2. Run-Time System Architecture Overview

The Mentat run-time system supports the macro data flow model. The run-time system presents a virtual macro data-flow machine to Mentat applications via the provision of five basic services: Mentat object management, token (message) matching, program graph support, guarded statement (predicate) support, and communication services. The Mentat run-time system architecture consists of a set of processors communicating through some interconnection network. Each processor has a complete copy of the run-time system.

The run-time system can be split into two parts, library routines that are linked with each Mentat object (user application), and the external scheduler and token matcher. The relationships between run-time system components are illustrated in Figures 1 & 2. Figure 1 shows the processor software architecture. Figure 2 shows the internal structure of a typical Mentat object. Each of the five services is briefly described below. In sections 4, 5, and 6, each will be described in detail.

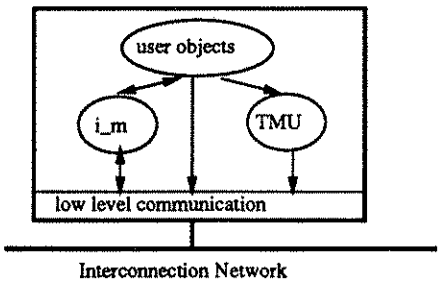

Figure 1.

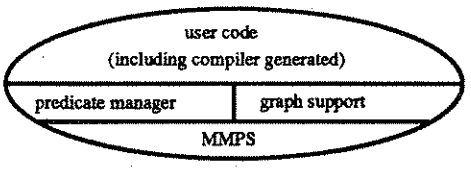

Figure 2.

Object management: Mentat object management consists of three sub-tasks: a scheduling service, a naming service, and an instantiation service. Although scheduling of Mentat objects to nodes is automatic, the algorithm is parameterizable and users may optionally supply hints to the scheduler to improve its decisions. The name service function permits objects to find instantiated objects of a particular class. The final service is to actually start an object once the location has been decided, and to give the object a unique name. All three of these tasks are performed by the *instantiation manager*, or *i_m*. (See Figure 1.)

Token matching: Mentat uses a data-driven model based on data-flow. Each node in a program graph may require more than one token (message) before it can begin computation. When a message arrives at a node, it must be determined whether the matching token(s) have arrived yet. When a token destination is unbound the message system delivers the message to the *token matching unit* (TMU). The TMUs (one per processor) cooperate with one another to match tokens for unbound computations. When a complete set of tokens is matched, the TMU forwards the tokens to an instance of the appropriate Mentat object for execution.

Graph support: Program graph support in Mentat is provided via the Mentat run-time library linked to each Mentat object. (See Figure 2.) Graph support consists of run-time construction of data flow subgraphs by detecting data dependence in the program, and the management of the interaction of an actor (node in a graph) with its corresponding graph.

Predicate management: The predicate management support is in essence an extension of message passing. The user (via the compiler) may specify a *predicate* to test or block upon. The predicates allow the user to specify the characteristics of the messages it is interested in receiving. All other messages are cached until needed. Predicates facilitate the implementation of sophisticated *select/accept* statements in which guards may depend not only on local variables and constants, but on the arguments to the functions as well.

Communication: Mentat communication is via message-passing. Note though, that the user does not see the message passing system; the compiler, the graph management routines, and the predicate management routines do. Basic message services are provided by **MMPS** [7], a portable, customizable, message-passing system.

3. Macro Data Flow Model

To better understand the run-time system, we first present a brief description of the model of computation used by Mentat, the *macro data flow model* [8-9], a medium-grain, data-driven model inspired by the data flow model [10-11]. A data flow program is a directed graph in which nodes are computation primitives called *actors*. *Tokens* carry data and control information along the arcs from actor to actor. When tokens are present on all incoming arcs, an actor is enabled and may execute. Thus a high degree of parallelism can be achieved naturally. Macro data flow has three principle differences from traditional data flow. First, the granularity of the actors is larger and under programmer control. This provides the flexibility to choose an appropriate degree of parallelism. Second, some actors can maintain state information between executions. These are called *persistent actors*. Persistent actors provide an effective way to model side effects and to reduce communication. Third, the structure of macro data-flow program graphs is not fixed at compile time. Instead, program graphs grow at run-time by elaborating actors into arbitrary subgraphs. In the macro data flow model, program graphs are represented by data structures called *futures* and *future lists*. A future represents a subgraph rooted at an

outgoing arc from an actor. A future list is a list of *futures*. An actor receives a future list with its arguments. The future list represents the subgraph rooted at the actor; each future in the future list corresponds to the subgraph rooted at an outgoing arc from the actor. The graph elaborations are completely local and do not affect any other actor or arc in the graph. The locality has important implications for distributed control. In Mentat, actors correspond to Mentat object member functions; and each Mentat object implements a set of macro data flow actors.

4. Run-Time Library Support

The Mentat run-time library is linked to all Mentat applications. The run-time library provides support for program graph construction via run-time data dependency detection, select/accept statement execution, and inter-object communication. Before discussing the library, we introduce the MPL and present an example of MPL code.

4.1. The MPL

The MPL was designed with four goals. First and foremost, the MPL would be object-oriented [13-14]. We would extend the usual notions of data and method encapsulation 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 would be unaware of whether the implementation of the member function is sequential or is parallel, i.e., whether its program graph is a single node, or whether it is a parallel graph. 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. Second, the language constructs should have a natural mapping to the macro data flow model; and the responsibility for performing the mapping should not be the programmer's. Third, the extensions should be applicable to a broad class of imperative languages. Fourth, the syntax and semantics of the extensions should follow the pattern set by the base language, maintaining its basic structure and philosophy whenever possible.

These goals were met in the MPL by extending the C++ language in five ways. The basic idea is to allow the programmer to specify those C++ classes that are of sufficient computational complexity to warrant parallel execution. This is accomplished using the **MENTAT** keyword in the class definition. Instances of Mentat classes are called Mentat objects. The programmer uses instances of Mentat classes like any other C++ class instance. The compiler generates code to construct and execute macro data flow graphs (data dependency graphs) in which the actors 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. All communication and synchronization is managed by the compiler. Of course, any one of the actors in a generated program graph may itself be transparently implemented in a similar manner by a macro data flow subgraph. Thus we obtain intra-object parallelism encapsulation; the caller only sees the member function invocation.

Briefly, the extensions to C++ are: **MENTAT** classes (both *persistent* and *regular*), the Mentat class member functions *create()* and *destroy()*, the *select/accept* guarded statements, and the *rtf()* (return to future) function.

Instances of Mentat objects are address space disjoint. All communication is via member function invocation. Parameter passing is by value. To instantiate and destroy instances of Mentat objects, we have added two new reserved member functions for all Mentat class objects: *create()* and *destroy()*. Create() allows the user to specify where the new instance is to be instantiated, e.g., on a different processor or on the same processor as another Mentat object.

The function *rtf()* is analogous to *return* in C++. Its purpose is to allow Mentat member functions (actors) to return a value to the successor nodes in the macro data-flow graph in which the member function appears. The "value" returned may be either a local variable or a local variable that corresponds to a subgraph that computes the result. These subgraphs are automatically generated as described below, and their use is transparent to the programmer.

The Mentat programming language has a *select/accept* statement that is similar to the ADA [15] select/accept. Guards may be assigned priority and are evaluated in the order of their priority. Like ADA guards, Mentat guards may contain local variables and constants. Mentat guards also may contain the formal parameters of the member function being guarded.

Example 1: Figure 3 illustrates both intra-object and inter-object parallelism encapsulation. Assume that the code fragment shown implements a node *alpha* that is embedded in some program graph. The elaboration of *alpha* into the subgraph shown in Figure 4 illustrates transparent parallelism from *alpha*'s perspective. Five instances of the Mentat class *bar* are defined, as are four integers. The class *bar* has two member functions, op1 and op2. Each member function requires some integer parameters and returns an integer. During the course of the execution of the code fragment, *alpha* will be replaced by one of two possible subgraphs depending on whether *expression(local_state)* evaluates to true or false.

*Not to be confused with Multilisp futures[12].

```
Mentat class bar {
public:
        int op1(int,int);
        int op2(int,int,int);
};
int w,x,y,z;
bar A,B,C,D,E;
w=A.op1(4,5);
if (expression(local_state))
        x=B.op1(w,5);
else    x=C.op1(w,10);
y=D.op1(7,w);
z=E.op2(y,w,x);
rtf(z);
```

Figure 3.

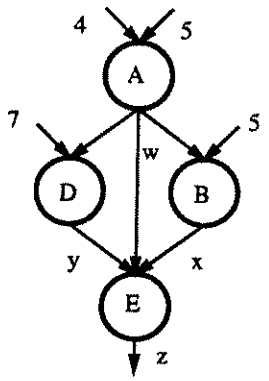

Figure 4.

If we assume that expression(local_state) is true, then the subgraph shown in Figure 4 results from the execution of the code fragment. The graph is constructed at run-time by noting that *B.op1* and *D.op1* depend on *w*, the output of *A.op1*, and that *E.op2* depends on *w*, *x*, and *y*. The final operation *rtf(z)* replaces the node executing this code with the subgraph just generated.

4.2. Graph Construction & Data-flow Detection

The MPL is supported in Mentat by translating MPL programs to C++ programs with embedded Mentat run-time library calls. This translation has three aspects. First, code is generated to perform run-time data flow detection. In general this requires expanding source statements into calls to the Mentat run-time library to construct program graphs and manage Mentat object invocation. Second, the select/accept statements are transformed into predicate manager calls, which in turn operate on the message queue associated with the Mentat object. Third, the implementation code of each Mentat class is broken out and separately compiled. Of these, the code generation for run-time data flow detection is the most interesting.

The execution path of a macro data-flow program is not known prior to execution [8-9]. Data flow must be detected at run-time and the data-flow graph constructed. Run-time data-flow detection is accomplished by detecting all uses of Mentat objects on the right-hand side of assignment statements and then monitoring where the left-hand side variable is used later. By carefully observing where the results of the Mentat object invocation are used, we can construct data-flow arcs from the node representing the Mentat object invocation to the nodes that use the result. To understand how this is done, we need some definitions.

Let *A* be a Mentat object with a member function *operation1(int,int)*. A *Mentat expression* is one in which the outermost function invocation is an invocation of a Mentat member function, e.g., the right-hand side of

x=A.operation1(5,4);.

A Mentat expression may be nested inside of another Mentat expression, e.g.,

x=A.operation1(5,A.operation1(4,4));.

The right-hand side of every *Mentat assignment statement* is a Mentat expression, e.g.,

x=A.operation(5,4);.

An instance of some type or class whose value is immediately available is an *actual value*. All variables and constants in standard C++ are considered actual values.

A *computation instance* contains the name of the Mentat object invoked, the *computation tag* that uniquely identifies the computation, a list of the arguments (either actual values or pointers to other computation instances that will provide the values), and a *successor* list. Computation instances are created by *invoke_fn()* when a Mentat object is invoked. *Invoke_fn()* marshalls arguments, creates the computation instance, and returns a computation instance pointer (a CIP). A computation instance contains sufficient information to acquire the actual value that is the result of the operation. We call the process of acquiring the actual value *resolving* the computation instance. Computation instances correspond to nodes in the data flow graph.

An element of the argument list of a computation instance may point either to a message containing the argument (marshalled by invoke_fn), or a pointer to another computation instance. If it points to a computation instance, then that computation instance corresponds to an immediate successor in the program graph being constructed. The successor list is a list of pointers to other computation instances that are dependent on the result of the computation instance. Each computation pointed to in the successor list will receive an identical copy of the computation's result. Sometimes the caller also requires a copy of the result; this is handled as well.

Program graphs are constructed by linking computation instances together via their argument lists and successor lists. The links correspond to the data dependencies observed at run-time.

A *result variable* (RV) is a variable that occurs on the left-hand side of a Mentat assignment statement. A result variable has a *delayed* value if the most recent assignment statement to it was a computation instance and the actual value for the computation instance has not been resolved. A result variable has an *actual* value if it has a value that may be used, i.e., it is not delayed. To detect data-flow at run-time we must monitor all uses of result variables, both on the left- and right-hand sides.

Each result variable *X* has a state designated *X.state* that is either delayed or actual. We define the *result variable set* (RVS) to be the set of all result variables that have a delayed value. Membership in RVS varies during the course of object execution. We define the *potential result variable set* (PRV) to be the set of all result variables. A variable may be a member of PRV and never be a member of RVS. Membership in PRV is determined at compile time.

The run-time library performs run-time data flow detection by maintaining a table of the addresses of delayed result variables called the *RV_TABLE*. The RV_TABLE contains the RVS, and is shown in figure 6. Each RV_TABLE entry contains the address of the result variable, and a pointer to a *computation instance*. If the address of a result variable is not in the RV_TABLE, then the result variable state is actual.

There are four functions of interest that operate on the RV_TABLE:

SET_ME((char) rv_address, CIP node),*

RV_DELETE((char) rv_address),*

RESOLVE((char) rv_address,int size)*, and

force().

SET_ME((char) rv_address, CIP node)* creates an entry in the RV_TABLE with a CIP value of *node* for the result variable pointed to by *rv_address*. If an entry already existed for *rv_address*, the associated computation instance is *decoupled*; since we know it can never be used again on the right-hand side, we can stop keeping track of it and execute it at our leisure.

Invoke_fn(arg_list) is an operator defined for Mentat objects. When *invoke_fn()* is called it creates a new computation instance for the computation, a new program graph node is created. *invoke_fn()* marshalls the arguments, both actual arguments (e.g., integers), and arguments that are computation instances. When an argument is a computation instance, that tells *invoke_fn()* that an arc should be added from the argument (a computation instance), to the new computation instance *invoke_fn()* is constructing.

RV_DELETE((char) rv_address)* deletes the RV_TABLE entry associated with *rv_address* if one exists. Before the entry is deleted, its associated computation instance is decoupled.

RESOLVE((char) rv_address,int size)* is called when the user program requires a value for a result variable. If an entry in the RV_TABLE exists for *rv_address*, RESOLVE *forces* the result, and blocks until the result is available. Once the result is available, RESOLVE places the result into the memory pointed to by *rv_address*. The *size* field indicates the maximum number of bytes to be copied into the result variable. If *size* is negative, it tells resolve that *rv_address* points to a pointer. In that case, RESOLVE *mallocs* enough space for the result, and sets the result variable to point to the allocated space. If the *rv_address* is not in the RV_TABLE, then RESOLVE does nothing.

Force() is used to begin the execution of any program graphs that have been constructed so far. It constructs the future lists from the program graphs and sends messages with the appropriate future lists to the appropriate objects.

The code in Example 1 has been expanded below in Figure 5 to show the library calls that are made to the run-time library. After each line of *C++* code is a brief explanation of what is happening. The casts of address to *char** have been removed for clarity. The RV_TABLE that results from execution of this code fragment is shown in figure 6. For more information on the MPL compiler and its transformations, see [2].

4.3. Predicate Management

The *predicate manager* (PM) sits between the user code and the token transport system. The PM receives all tokens destined for an object and returns matched sets of tokens to the user code. In the simple case the matched set returned is the first set of arguments to a member function invocation that have all arrived. In general, the user may have specified *guards* that must evaluate to *TRUE* before the corresponding member function can be invoked. Guards may be functions of the local state of the object and the contents of the messages. Guards may also be prioritized. The job of the PM is therefore twofold. First, it matches the tokens by computation tags. Second, it supports the MPL's *select/accept* guarded statement semantics and permits the user (really the compiler) to selectively accept only those messages, and matched message sets that the user wants. The user accomplishes this by blocking on (or testing) a *predicate*. The implementation of each of the two subtasks is described below.

The matching process is accomplished by first collecting together complete sets of tokens that, in the absence of guards, would enable an actor to fire. When a message arrives at a Mentat object, the destination field contains not only the name of the object, but several other fields as well. These fields are the *operation#*, the

```
(*SET_ME((&w)))=A.invoke_fn(101,2,
                ICON_TO_ARG(4),
                ICON_TO_ARG(5));
/* Set up an RV_TABLE entry for w, pointing to
  invocation of A. A has two integer constant
  arguments, they are marshaled. */
if (local expression)
(*SET_ME((&x)))=B.invoke_fn(101,2,
                PRV_TO_ARG((&w),4,0),
                ICON_TO_ARG(5));
/* B has two arguments, a potential
  result variable w, and an integer constant.
  The PRV is delayed, so an arc is created. */
else
(*SET_ME((&x)))=C.invoke_fn(101,2,
                PRV_TO_ARG((&w),4,0),
                ICON_TO_ARG(10));
(*SET_ME((&y)))=D.invoke_fn(101,2,
                ICON_TO_ARG(7),
                PRV_TO_ARG((&w),4,0));
(*SET_ME((&z)))=E.invoke_fn(102,3,
                PRV_TO_ARG((&y),4,0),
                PRV_TO_ARG((&w),4,0),
                PRV_TO_ARG((&x),4,0));
/* Note that all three arguments are
  PRVs, three arcs are added.     */
rtf(PRV_TO_ARG((&z),4,0));
```

Figure 5. Expanded code of example 1.

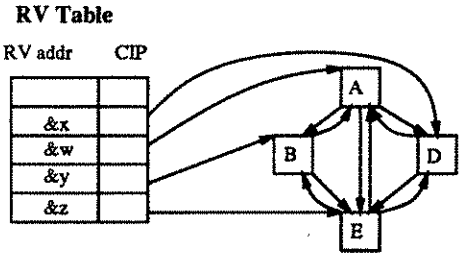

Figure 6.
RV_TABLE and program graph for example 1.

argument#, and the *computation tag*. The *operation#* specifies which member function is to be invoked. The *argument#* specifies which argument the message supplies. The computation tag is a unique identifier that specifies a particular invocation of the object.

Before a member function can be executed, all of its arguments must have arrived, i.e. there must exist a set of messages with the same *operation#* and with the same computation tag such that all of the arguments are present. The process of finding such a set is called *token matching*. Once a set of tokens has been matched we say that the operation is *enabled* and may *fire*.

The second part of the PM's responsibility is to implement the MPL's *select/accept* statements. This is accomplished using *predicates*. A predicate is a table, each entry of which has five fields, an *operation#*, a priority, the number of arguments expected, a pointer to a *guard function* (possibly null), and a pointer to the head of the guard function argument list (possibly null).

A predicate corresponds to a set of member functions to be *accepted*, and their associated guards and priorities, that the user may wish to block on or test. There is usually a one to one correspondence between predicates and *select/accept* statements. The guard is a boolean function that is applied against matching sets of tokens that have an appropriate *operation#*. If the guard function evaluates to TRUE the corresponding member function is eligible for execution. If the guard function evaluates to FALSE, the tokens did not satisfy the guard and the matched tokens are not yet eligible for execution under that guard. They may, however, match under another guard or at some later time.

The priority field of the predicate table entries are used to determine the order in which guards are evaluated. This gives the user (via the compiler) the ability to schedule the order in which requests (member function invocations) are satisfied. In the event that two entries have the same priority, the order of evaluation is unspecified (non-deterministic).

There are four member functions defined for predicates, *enable operation*, *disable operation*, *block*, and *test*. *Enable* and *disable* set up and disable predicate table entries respectively. *Block* tests the guards in order of priority until a guard evaluates to TRUE or until there are no more matched tokens to test. If a guard evaluates to TRUE the predicate table entry number and pointers to the matched messages are returned to the caller. If none evaluates to true, *block* blocks on message reception, and tries again once a new message has arrived. *Test* is similar to *block*, except that the guards are only tested, the message pointers are not returned, and *test* does not block on failure.

The MPL compiler transforms *select/accept* statements into: 1), the construction of appropriate guard functions and a predicate, 2), the enabling of the member functions corresponding to the *accepts* with the appropriate guard pointers and priorities, 3), a call to *block*, and, 4), a *switch* statement on the result of *block*, with the appropriate code in the cases.

4.4. Communication

Mentat communication is via message passing. Basic message services are provided by **MMPS** [7]. The run-time system provides guaranteed delivery of messages of arbitrary size from one Mentat object to another. Message reception is handled by the predicate manager discussed above. Some messages may be destined for

unbound objects. Unbound object destinations are used when it is not important which instance of an object class receives the message, only that an instance receives it. Unbound messages are sent to the TMU (described below).

5. Object Management

Object management is performed by the *i_m*. Object management consists of three subtasks: a scheduling service, a naming service, and an instantiation service. Below we will concentrate on the most interesting, scheduling.

The scheduler is built upon the scheduling research of Eager, Lazowska and Zahorjan [16], who developed a *sender-initiated adaptive load sharing* model for homogeneous distributed systems. It describes the way in which the current load in the system is distributed among its components using system state information. Mentat, however, was designed to work on system architectures and topologies in which processors do not necessarily have equal performance characteristics. Furthermore, the Mentat programming language (MPL) provides information on process characteristics that can be used by the scheduler to improve its decisions. This makes the pure Eager, Lazowska and Zahorjan model insufficient. Our model accommodates these environmental differences.

Our model deals exclusively with global scheduling, and is only concerned with deciding where to execute a process. Local scheduling, which assigns processes to time-slices of a single processor, is left to the host operating system of the processor to which the process is ultimately allocated. A *task scheduling* mechanism is used because, in the macro data flow model, each actor, or object, is treated as an independent task by the task scheduling mechanism.

We represent distributed systems as collections of not necessarily identical nodes, each consisting of a single processor. The nodes are connected by a local area broadcast channel or by other communications medium. There is one exact copy of the scheduler in each node, and the schedulers perform an independent decentralized function. Our scheduler has the following characteristics:

1. ***Distributed:*** The scheduling decisions are distributed and independent among the systems.
2. ***Load Sharing:*** The *i_m* transparently distributes the workload by transferring tasks from nodes that are heavily loaded to nodes that are lightly loaded.
3. ***Adaptive:*** The *i_m* employs information on the current system state in making transfer decisions.
4. ***Sender-Initiated:*** Congested nodes search for lightly loaded nodes to which work may be transferred.
5. ***Static Assignment:*** Each task remains on the node to which it is assigned until completion; there is no task migration.
6. ***Stable:*** A task can only be transferred a fixed number of times between the nodes of the system. Transfers can only occur before task execution. Hence, processor thrashing is avoided.

The *i_m* divides the scheduling decision into two components: the *transfer policy*, which determines whether to process a task locally or remotely, and the *location policy*, that determines the processor to which a task is sent. The *i_m* can base its transfer decisions not only in the run queue length of the processors, but also on other information about the system state such as available memory and CPU utilization. The *i_m* uses a threshold transfer policy that supports several different location policy algorithms with a static transfer limit. There are currently three algorithms for the location policy process. They are *random*, *round-robin* and *best-most-recently*. The division of the scheduling decision into two components makes it easy to incorporate additional algorithms into the location policy component. The static transfer limit makes our model *stable* and prevents *processor thrashing*.

The *i_m* may be configured for different architectures. A configuration is based on the state information that the host operating system supplies to the scheduler. Some systems offer easy and quick access to their state information (SunOS [5]), whereas others (Intel's NX/2 [4]) do not provide an easy way to collect this information. Thus, different forms of state information are used in different architectures. The *i_m* supports different system topologies and processor power heterogeneities by the creation of *logical sub-networks* of homogeneous processors. These logical sub-networks also give a separation between physical sub-networks.

Our model uses the *location hints* specified by the programmer to improve the scheduling decisions. There are currently three types of location hints which can be specified using MPL. They are: *CO-LOCATE*, *DISJOINT* and *HIGH-COMPUTATION-RATIO*. CO-LOCATE tells the scheduler to locate the object being instantiated close enough to another object so that communication between the two is inexpensive. DISJOINT tells the scheduler that an object should be instantiated far away from any object in a given list because it will usually be executed in parallel with those in the list. HIGH-COMPUTATION-RATIO tells the scheduler that the object to be instantiated has a particularly high computation ratio. The scheduler uses this information to ensure that it is placed on a lightly loaded or powerful processor, even if the processor is very far away.

Due to space restrictions a full discussion of the location policies and parameters is not possible. The scheduler, and our scheduling results, are the topic of a future paper. For more information on the *i_m* and the on performance of different location and transfer policies using various parameters and work loads see [17].

6. Token Matching

When a member function of a regular object (or an uninstantiated object) is invoked the run-time system is responsible for matching the tokens when they become available and for instantiating an instance of the class to execute the member function. Recall that regular object member functions are pure functions; thus all instances of regular objects are equivalent, and we may use any instance to execute the member function.

Token matching for uninstantiated objects is complicated by the distributed nature of the matching problem: how can we know where the tokens we are trying to match reside, or whether all of the tokens to be matched have even been generated yet? Our solution to this problem is to send all tokens that are destined for uninstantiated objects (regular objects) to the local *token matching unit* (TMU). Each host in a Mentat system has a TMU that is responsible for collecting all of the tokens together in one place and then executing the member function.

To collect the tokens together, the TMUs must cooperate. One of the TMUs is specified as the *coordinator* of each computation. The coordinator is that TMU which receives the token containing the last argument of the member function. ("Last" here refers to the order in which the formal parameters are defined, not the order in which the arguments are generated.) All other TMUs that receive tokens containing the other arguments are known as assistants. A TMU may be both a coordinator and an assistant for different computations.

The coordinator is responsible for gathering all of the tokens (messages) for a particular computation. If the member function requires only one argument, this is trivial. If there is more than one argument, the coordinator sends out *probes* to its fellow TMUs requesting all matching tokens they may have. When the coordinator has received all of the required tokens, it first requests an instance of the desired class from the *i_m* and then sends messages to other TMUs canceling the request for matching tokens. When the *i_m* provides an instance name, all of the tokens are forwarded to the named object. This object performs the requested computation. The problem with this solution is the message complexity, $O(2n)$, where n is the number of hosts in the system. A more efficient algorithm based on hashing is being investigated.

7. Performance Results

7.1. 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, computation instance operations, and predicate operations must all incur little cost, or overhead will swamp the gains from parallelism. If the cost to construct program graphs dynamically is too high, then static graph methods, with the resulting increase in the number of actors, must be used. In Tables T-1 and T-2 below, the execution times for the primitive operations are given for a Sun 3/60 and a node on the Intel iPSC/2 respectively. The time to execute a null RPC is also given, as it represents the time to construct a simple graph, send the parameter, schedule and execute the RPC, and reply to the call. Note that the message transport cost includes the host operating system scheduling and task switch overhead. For more information on the communication times see [7].

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

Function	Time
Create/Destroy computation instance	65μS
Add Arc	32μS
Create/Destroy Message	260μS
Per node overhead	357μS
Transport Message (Send)	9.8mS
Null RPC	22mS

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

Function	Time
Create/Destroy computation instance	56μS
Add Arc	32μS
Create/Destroy Message	94μS
Per node overhead	357μS
Transport Message (Send)	800μS
Null RPC	2.8mS

The largest cost operation is communication. The use of control actors (as required for static graphs) would dramatically increase the number of messages. By constructing the graphs at run-time, thereby avoiding the use of control actors, significant savings in time can be realized.

7.2. Applications

Nice computation and programming models aside, the bottom line for parallel systems is performance. As of this writing we have implemented the Mentat run-time system on two different architectures: a network of Sun workstations, and a sixteen node Intel iPSC/2 Hypercube. This version is our first and has not been optimized. Rather than concentrating on speed, our efforts have been spent on getting it right. We believe that substantial performance improvements are possible as the components of Mentat become faster, reducing overhead. 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. 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.

7.3. Execution Environment

The data were collected in two different environments: a network of Sun workstations and a Hypercube. The network of Suns consists of 10 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 run-times used to compute the speed-ups for the Suns were obtained late at night.

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 VM, coupled with the amount of memory consumed by the OS, 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 up into n pieces $A_1 \ldots A_n$, 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.

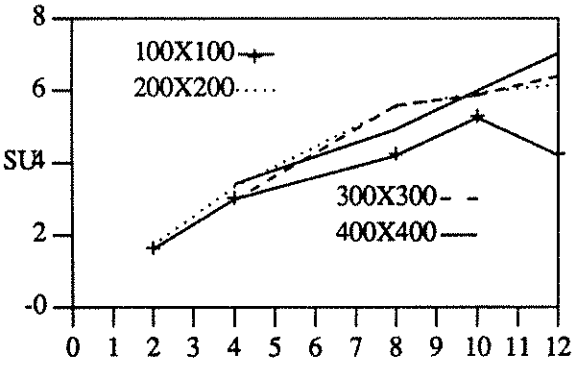

Figure 7. Sun network matrix multiply

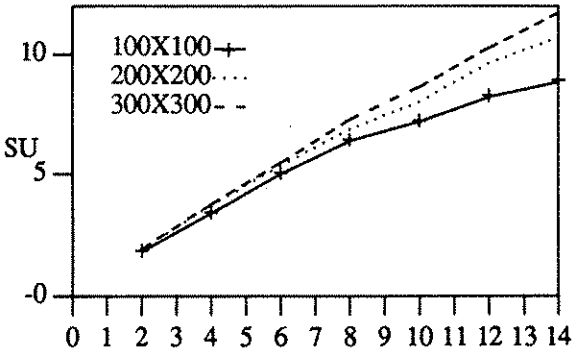

Figure 8. Hypercube matrix multiply

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, call the reduce operator for each *sblock* using the partial pivot calculated at the end of the last iteration. The reduce operation of the *sblocks* both reduces the *sblock* by the vector and 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 & 10 are compared to the results for matrix multiply.

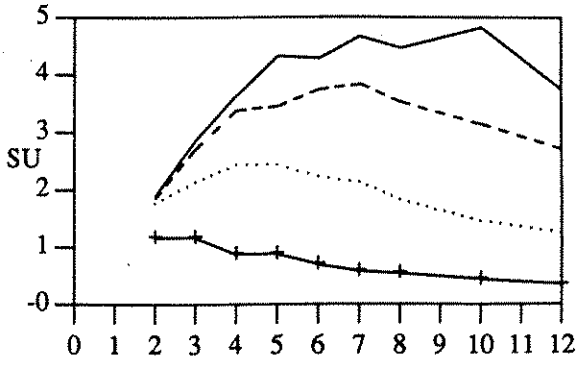

Figure 9. Sun Gaussian

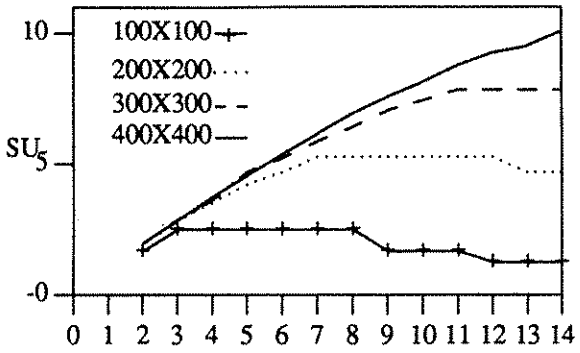

Figure 10. Hypercube Gaussian

8. Summary

The goal of Mentat is to provide easy-to-use parallelism to non computer-scientists. Mentat meets the goals by providing an easy-to-use, parallel, object-oriented programming language, the MPL, and by providing run-time support for the MPL on a wide variety of architectures. This paper describes the run-time system architecture. Mentat has been ported to the Intel Hypercube, and a network of BSD Unix hosts. Performance figures for both systems are encouraging. A port to Mach is underway, with completion expected in the summer of 1990. Future work on the run-time system will concentrate on optimizing the run-time system to improve

application performance, and on fault-tolerance. We are currently implementing two real-world applications in Mentat to test our "easy-to-use" claim. Results are expected in the fall of 1990.

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 all of her comments and assistance. I would also like to thank the Institute for Parallel Computation for the use of their Intel Hypercube.

References

[1] 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.

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

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

[4] Intel Corporation, "iPSC/2 USER'S GUIDE," Intel Scientific Computers, Beaverton, OR, March 1988.

[5] Sun Microsystems Inc., SunOS Users Manual, Mountain View, CA, 1988.

[6] S. J. Leffer, R. S. Fabry, and W. N. Joy,"4.2BSD Interprocess Communication Primer," Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, July 1983.

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

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

[9] 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.

[10] J. Dennis, "First Version of a Data Flow Procedure Language," MIT TR-673, May, 1975.

[11] A. H. Veen, "Dataflow Machine Architecture," *ACM Computing Surveys*, pp. 365-396, vol. 18, no. 4, December, 1986.

[12] R. H. Halstead Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, pp. 501-538, vol. 7, no. 4, October, 1985.

[13] 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.

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

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

[16] D. L. Eager, E. D. Lazowska and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Transactions on Software Engineering*, vol. 12, pp. 662-675, May 1986.

[17] V. E. Vivas and A. S. Grimshaw, "Design and Implementation of a Distributed Schedler for Mentat," Technical Report in progress, Department of Computer Science, University of Virginia, Charlottesville, VA.