Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing

Andrew S. Grimshaw,Jon B. Weissman,W. Timothy Strayer

Technical Report No. CS-93-40
July 14, 1993

This work partially funded by NSF grants ASC-9201822 and CDA-8922545-01, and NASA NGT-50970.

Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing[1]

Andrew S. Grimshaw
Jon B. Weissman
W. Timothy Strayer

Abstract

The object-oriented paradigm is a powerful tool for managing software complexity. A key question when the paradigm is applied to parallel computing is whether the associated overhead is so large as to defeat the high-performance objectives that motivate parallel computing. We show that high-performance and dynamic object-oriented parallel processing are not mutually exclusive. Our vehicle is Mentat, a portable, object-oriented parallel processing system developed at the University of Virginia. We present a brief overview of the Mentat Programming Language, a detailed description of the run-time system that supports the language, and the performance of run-time system primitives on two disparate platforms, a Sun SparcStation 2 and the Intel iPSC/860 Gamma. The results, combined with application results presented elsewhere, lead us to the conclusion that high-performance and dynamic, object-oriented parallel processing are not mutually exclusive.

Keywords: object-oriented, parallel processing

1. Introduction

The object-oriented paradigm has proven to be a powerful tool for managing complexity in the development of software for sequential computers, and its power is being exploited in the more complex domain of parallel software as well [3][4][6][14][22][23]. In parallel systems, where performance is a key objective, the issue of run-time support must be addressed, particularly for distributed memory MIMD machines. For years the object-oriented approach has had a reputation for high run-time overhead and poor performance. Recent sequential languages such as C++ have avoided the performance problem associated with earlier object-oriented languages. Given that high performance is the *raison d'etre* of parallel computing, performance cannot be allowed to suffer due to run-time overhead. Therefore, if the object-oriented approach is to be successfully applied to parallel systems, high-performance, efficient run-time support must be provided.

Mentat is an object-oriented parallel processing system designed to simplify the task of writing portable, high-performance, parallel applications software [12][14][15][26]. The fundamental objectives of Mentat are to (1) provide easy-to-use parallelism, (2) facilitate the portability of applications across a wide

[1] This work is partially funded by NSF grants ASC-9201822 and CDA-8922545-01, and NASA NGT-50970.

range of platforms, and (3) achieve high performance. The first two objectives are addressed through Mentat's underlying object-oriented approach: high-level abstractions mask the complex aspects of parallel programming, including communication, synchronization, and scheduling. The question is whether Mentat — or any parallel processing system — can meet the first two objectives and not sacrifice the third. How Mentat dynamically supports its object-oriented programming language, and how it does this efficiently, is the topic of this discussion.

Mentat has two primary components: the Mentat Programming Language (MPL) and the Mentat run-time system. The MPL is an object-oriented programming language based on C++. The granule of computation is the class member function. The programmer is responsible for identifying those object classes whose member functions are sufficiently complexity to allow efficient parallel execution. Instances of these special classes are used exactly like C++ classes.

The compiler and run-time system work together to ensure that the data and control dependencies between Mentat class instances are automatically detected and managed without programmer intervention. The underlying assumption is that the programmer's strength is in making decisions about granularity and partitioning, while the compiler together with the run-time system can better manage communication, synchronization, and scheduling. This simplifies the task of writing parallel programs.

The high-performance aspects of Mentat have been demonstrated in [12][15]. For a general overview of Mentat and of the Mentat parallel processing philosophy see [14]. In this paper we present the Mentat run-time system which supports the Mentat Programming Language. Our objectives are twofold: first to describe the inner workings of the run-time system, its software architecture, and the structure and interaction of its components, and second, to demonstrate that dynamic object-oriented parallelism can be efficiently exploited. We begin by briefly describing the Mentat Programming Language and the macro data flow model of computation. Macro data flow is the model used by Mentat, and implemented by the run-time system. The run-time system architecture is our next focus, starting with the virtual machine structure, a description of the services provided, and their design. We follow that with a sketch of the unique implementation aspects and performance of two of the run-time system implementations, on the Sun SparcStation 2 and the Intel iPSC/860 Gamma.

2. Mentat Programming Language

Rather than invent a new language for writing parallel programs, the Mentat Programming Language (MPL) is an extension of the object-oriented language C++. The extensions allow the

programmer to provide granularity and data decomposition information to the compiler and run-time system. In particular, the keyword “mentat” in the class definition modifies the semantics of a class to include decoupling a member function’s execution from the sequential flow of the calls to the function. This allows the programmer to specify those C++ classes that are sufficiently complex to warrant parallel execution.

Instances of Mentat classes are called *Mentat objects*. The compiler generates code to dynamically construct and execute “macro” dataflow graphs at run-time where the actors are Mentat object member function invocations and the arcs are the data dependencies found in the program. We call this *inter-object parallelism* because parallelism opportunities *between* objects are being exploited. All communication, argument marshalling, and synchronization are managed by the compiler acting in concert with the run-time system. The actors in a generated program graph may themselves be transparently implemented in a similar manner by a macro dataflow subgraph. This is called *intra-object parallelism* encapsulation; the caller only sees the member function invocation and is unaware that the function is implemented in parallel.

2.1. The Macro Dataflow Model of Computation

Macro dataflow is Mentat’s underlying model of computation. The macro dataflow (MDF [11]) model is a medium grain, data-driven computation model inspired by dataflow[1][9][24][28]. Recall that in dataflow, programs are directed graphs where the vertices are computational primitives (e.g., add, subtract, compare, etc.) called *actors*, the edges, or *arcs*, model data dependencies, and *tokens* carry data along the edges between the actors. Dataflow is data-driven in that programs are self-synchronized by data motion. An actor may only execute when all of the required data, in the form of tokens, has arrived.

Macro dataflow differs from traditional dataflow in three ways. First, the computation granularity is larger than in traditional data flow [2][5][7]. Actors are high-level functions such as matrix-multiply specified in a high-level language, not primitive operations (e.g., add). Second, some actors, called *persistent actors*, may maintain state between invocations. Sets of persistent actors may share the same state. The actors that share the same state are executed in mutual exclusion, in a monitor-like fashion. Third, programs graphs are not fixed at compile-time; instead, program graphs are constructed at run-time by observing the data dependencies as execution unfolds. Thus, program graphs in MDF are dynamic and unknown at compile-time.

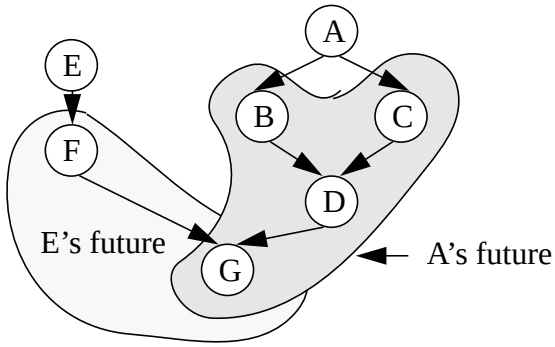


Figure 1 . A macro data flow subgraph. The future of the actor A is shown.

Program graphs are represented at run-time in MDF using *futures.*[2] A future represents the future of the computation with respect to a particular actor at a particular instant in time. For example, consider the program graph fragment of Figure 1. *A*'s future is shown by the shaded area enclosing the actors (B, C, D, G). The future of *A* at this point in time includes all computations that are data-dependent on the result of the computation that *A* performs. Although actors *E* and *F* are in the same program graph as *A*, they are not in *A*'s future, nor is *A* in their future.

When an actor such as *A* receives its tokens, a future is attached. When the actor completes, one of two things happens: the actor returns a value that is transmitted to each direct descendent in its future, or the actor elaborates itself into a subgraph. In either case, modifications need to be made to the program graph, either to reflect the completion of the actor or to include the new subgraph. Because the modifications require changing *A*'s future only, the modifications can be made locally. Other processors, such as those executing *E* or *F*, need not be notified.

Suppose that *A* returns the value 5. Since *A* has two output arcs, *A*'s future is broken into two futures which, along with the value 5, are forwarded to *B* and *C*. The new state is shown in Figure 2. *B* and *C* are now enabled and may execute since they have a token (value) on each input arc.

Alternatively, *A* may elaborate itself into an arbitrary subgraph. Suppose that *A* elaborates into the subgraph shown in Figure 3(a). The new state is shown in Figure 3(b). In this case the graph has grown, rather than contracted as in Figure 2. The point is that in both cases only *A*'s future needs modification; neither *E*, nor any other actor need be notified of the change.

[2.] MDF futures should not be confused with Multilisp futures[20].

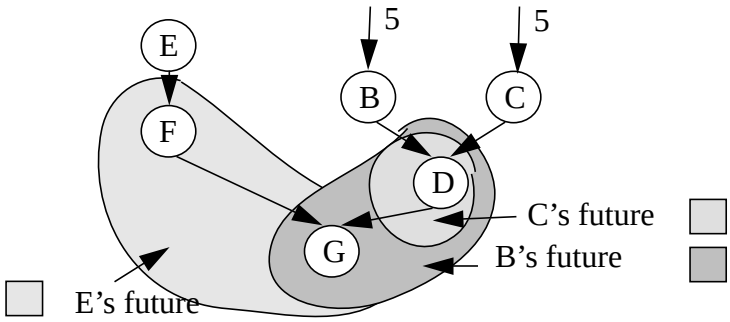
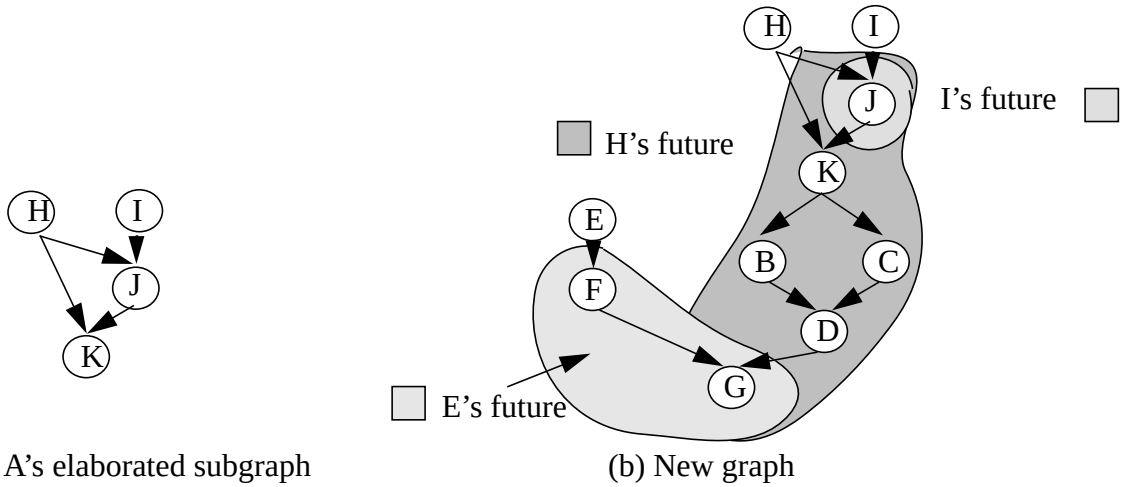


Figure 2 . New futures after actor A from Figure 1 returns a value. When two (or more) actors' futures come together at a third actor, e.g., B and C at D, only one future contains the graph beyond the third actor.



(a) A's elaborated subgraph (b) New graph

Figure 3 . Actor elaboration. The actor A from Figure 1 mat elaborate into the subgraph of (a).The resulting graph is shown in (b).

2.2. Mentat Classes

The most important extension to C++ is the keyword "mentat" as a prefix to class definitions, as shown on line 1 of Figure 4. This keyword indicates to the compiler that the member functions of the class are computationally expensive enough to be worth doing in parallel. Mentat classes are defined to be either *regular* or *persistent*. The distinction reflects the two different types of actors in MDF. Regular Mentat classes are stateless, and their member functions can be thought of as pure functions in the sense that they maintain no state information between invocations. As a consequence, the run-time system may instantiate

a new instance of a regular Mentat object to service each invocation of a member function from that class, even while other instances of the same function already exist.

Persistent Mentat classes, on the other hand, do maintain state information between member function invocation. Since state must be maintained, each member function invocation on a persistent Mentat object is served by the same instance of the object.

Each Mentat object possesses a unique name, an address space, and a single thread of control. Because Mentat objects are address space-disjoint, all communication is via member function invocation. Because Mentat objects have a single thread of control, they have monitor-like properties. In particular, only one member function may be executing at a time on a particular persistent object. Thus, there are no races on contained variables.

Variables whose classes are Mentat classes are analogous to variables that are pointers. They are not an instance of the class; rather, they name or point to an instance. We call these variables *Mentat variables*. As with pointers, Mentat variables are initially *unbound* (they do not name an instance) and must be explicitly *bound*. A bound Mentat variable names a specific Mentat object. Unlike pointers, when an unbound Mentat variable is used and a member function is invoked, it is not an error. Instead, if the class is a regular Mentat class, the underlying system instantiates a new Mentat object to service the member function invocation. The Mentat variable is not bound to the created instance.

2.2.1. Member Function Invocation

Member function invocation on Mentat objects is syntactically the same as for C++ objects. Semantically, however, there are three important differences. First, Mentat member function invocations are non-blocking, providing parallel execution of member functions when data dependencies permit. Second, each invocation of a regular Mentat object member function causes the instantiation of a new object to service the request. This, combined with non-blocking invocation, means that many instances of a regular class member function can be executing concurrently. Finally, Mentat member functions are always call-by-value because the model assumes distributed memory. All parameters are physically copied to the destination object. Similarly,

```
1  mentat class bar {
2  // private member functions and variables
3  public:
4     int op1(int,int);
5     int op2(int, int);
6  };
```

Figure 4 . A Mentat class definition. Without the keyword “mentat”, it is a legitimate C++ class definition.

return values are by-value. Pointers and references may be used as formal parameters and as results, but the effect is that the memory object to which the pointer points is copied. Variable size arguments are supported as well, since they facilitate the writing of library classes such as matrix algebra classes.

2.2.2. The Return-to-Future Mechanism

Mentat member functions use the return to future (`rtf`()) to return values. The value returned is forwarded to all Mentat object member function invocations that are data-dependent on the result, and to the caller if necessary. For example, in Figure 2, A performed `rtf(5)`. If the caller does not use the value, a copy is not returned.

While there are many similarities between the C `return` and the MPL `rtf`, they differ in three significant ways. First, a `return` returns data to the caller. An `rtf` may or may not return data to the caller depending on the data dependencies of the program. If the caller does not use the result locally, then the caller does not receive a copy. This reduces communication overhead. Second, a C `return` signals 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`. This is analogous to send-ahead in message passing systems. 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, as in Figure 3. Indeed, the result may be computed by a parallel subgraph obtained by detecting inter-object parallelism.

2.2.3. The mselect/maccept Statement

The MPL `mselect/maccept`[3] statement is modeled on the Ada `select/accept`. The programmer may specify which member functions are candidates for execution by including them in the `mselect`. As in Ada, the entries may be protected with a guard that must evaluate to true at run-time in order for the member function to be a candidate for execution. In addition to the guards, `maccept`

[3] We use mselect/maccept rather than select/accept because the Unix file system interface includes a function select() that permits selective blocking on I/O.

statements may be given a priority. The member functions are accepted in priority order. Unlike Ada there is no delay option since the underlying computation model has no concept of time. Instead, an `else` clause may be specified that will be taken if none of the guarded statements are enabled.

3. The Mentat Run-Time System

The Mentat run-time system efficiently supports parallel object-oriented computation on top of a data-driven, message-passing model. It supports more than just method invocation by remote procedure call; in addition, 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 RTS constructs program graphs, manages communication and synchronization, performs object instantiation and scheduling, and allows selective message reception to support an ADA-like select/accept semantics.

The run-time system supports MPL's model of computation, where each Mentat class member function corresponds to an MDF actor, and each formal parameter corresponds to an incoming arc for that actor. Tokens correspond to the actual parameters of the member function invocation. When all of the tokens have arrived and an actor is enabled, all of the actual arguments are available and the member function may execute.

MPL's model of computation is executed on a virtual macro dataflow machine (VMDFM) which provides the services used by the compiled code. This virtual machine also abstracts the platform-specific details so that the MPL code is portable across various MIMD architectures. The run-time system is currently running on several systems: the Intel iPSC/2 and iPSC/860 using NX/2 [21], and networks of Sun, Silicon Graphics, and IBM RS 6000 workstations using UDP packets and Unix sockets.

3.1. The Virtual Machine

The virtual macro dataflow machine, as shown in Figure 5, provides a service interface to the compiled MPL code. The services are divided into two groups, those library routines that are linked with application objects, and those that are provided by independent, daemon, Mentat objects. The library services include communication, dataflow detection, and guard and predicate evaluation. Object scheduling and instantiation is provided by instantiation managers (IM's), and unbound token matching by the token matching units (TMU's). The IM's and TMU's are daemons.

Internally, the virtual machine uses a classic layered approach. The top level provides an interface through which the compiler-generated code interacts with the virtual machine. The interface in turn is

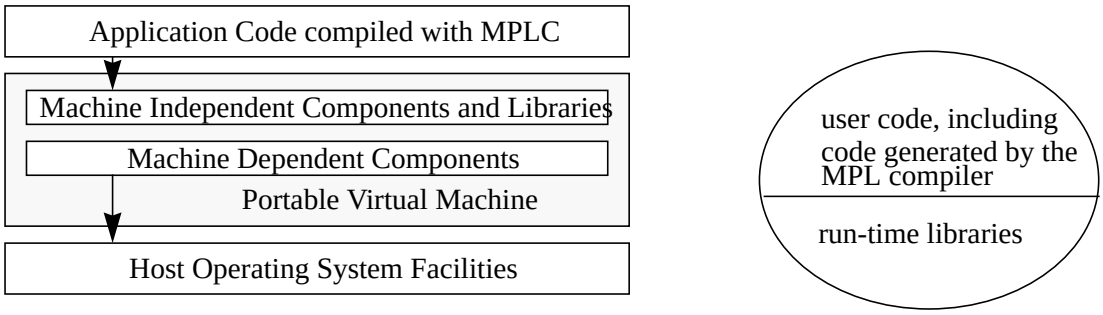


Figure 5 —Virtual Macro DataFlow Machine. (a) The VMDFM uses a layered approach to achieve portability. (b) User objects consist of user code and calls to run-time library routines that implement the virtual machine.

implemented as machine-independent modules. At the bottom level are the architecture- and operating system-specific modules where all platform-specific code has been isolated. The platform-specific modules include the communication system MMPS (the Modular Message Passing System [18]), and the object loader used by the instantiation manager. By isolating platform-dependent code we simplify both software maintenance and the task of porting Mentat to new platforms.

A second view of the RTS, shown in Figure 6, represents Mentat application objects and the two

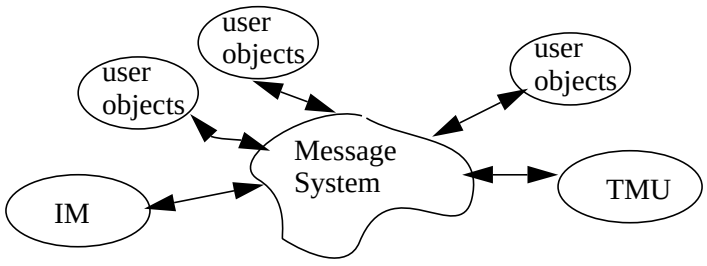


Figure 6 —Highest level of abstraction. Objects communicate via a reliable message passing system. There is no shared memory, nor is a there notion of processors.

daemon RTS objects as a set of address space-disjoint processes communicating via a reliable message passing system. Each object (including the daemons) has a unique name, an address space, and a thread of control. Communication between objects is solely through the message passing system - there is no shared memory. (Objects may communicate via the file system - a form of slow, usually incoherent, shared memory. We make no attempt to synchronize access to the file system.)

```
1  class mentat_object {
2     object_name i_name;
3  public:
4     CIP invoke_fn(int fn_number,int arg_count,arg_struct ...);
5     // invokde_fn is used to communicate with back-ends
6     void create(); // instantiate new back-end
7     void destroy();// destroy the back-end
8  };
```

Figure 7 Partial interface of the front-end `mentat_object` class.

3.2. System Services

There are five basic services provided by the run-time system: (1) object naming and basic communication, (2) data flow detection, (3) predicate management, (4) unbound token matching, and (5) object scheduling and instantiation. As discussed earlier, the first three are implemented by libraries linked with applications code and the latter two are implemented by daemons.

The run-time libraries are linked to each Mentat application and to the daemon objects. The library interface is the virtual machine used by the MPL compiler. Naturally the interface is object-oriented, and consists of a set of class definitions. Instances of these classes are manipulated by the compiler. The most important classes are `mentat_object`, `mentat_message`, `computation_instance`, and `predicate_manager`.

mentat_object

The RTS implementation of Mentat objects consists of two components, (1) the front-end class `mentat_object`, that contains the name of a Mentat object (process) and is the handle with which operations on the Mentat object are performed, and (2) a back-end server object process that contains the Mentat object's state and performs the member functions (user objects in Figure 5). Member function invocation involves using the front-end as a surrogate for the back-end server object. The front-end `mentat_object`s are essentially object names and a set of member functions used to communicate with the back-end server. The compiler generates code to manipulate `mentat_object`s and the server loops that implement the back-ends. The three member functions of interest are shown in Figure 7.

mentat_message

A `mentat_message` is a message that contains the destination object name, the destination function number, an argument number, a computation tag, a future, and the data. Mentat messages have the

property that if the destination object name is not bound, i.e., it does not name a specific instance, the message will be sent to a TMU.

computation_instance

The RTS keeps track of Mentat object member function invocations at run-time using c*omputation instances*, which correspond to nodes in a MDF program graph. They contains the name of the Mentat object invoked, the number of the invoked member function, the *computation tag* that uniquely identifies the computation, a list of the arguments (either values or pointers to other computation instances that will provide the values), and a successor list (also computation instances). A computation instance contains sufficient information to acquire the value that is the result of the operation. A CIP is a computation instance pointer.

predicate_manager

A `predicate_manager` specifies a predicate which is a set of member functions that are candidates for execution. The compiler constructs `predicate_managers` for each `mselect/maccept` in the program. The `predicate_manager` class implements the token matching function for bound Mentat objects.

3.2.1. Naming and Basic Communication

Naming and basic communication services are provided by the Modular Message Passing System (MMPS)[18]. MMPS provides C++ class-based naming and message passing services. The services provided are message construction, synchronous and asynchronous send, and blocking and non-blocking receive.

A MMPS-name is an address that names a specific object (process) on a specific processor. It contains sufficient information to allow the implementation to communicate with the object using the underlying host operating system's IPC primitives. For example, on UNIX systems using UDP datagrams, a MMPS name contains a 30 character hostname (IP format) and a port number. On the Intel iPSC/2 and iPSC/860, a MMPS name contains an integer host identifier and an integer process identifier.

The use of an address in the MMPS name means that named objects may not migrate — a restriction that we have not found burdensome to date. However, the use of IP host names in the socket implementation of MMPS names was a poor choice. MMPS names are used in many other data structures, often several at a time. The inclusion of the 30 byte host name significantly increases message lengths, leading to higher

```
1  bar A,B,C;
2  int w,x,y;
3  w = A.op1(4,5);
4  x = B.op1(6,7);
5  y = C.op1(w,x);
6  rtf(y);
```

(a) Draw an arc from A.op1() and B.op1() to C.op1().

```
1  bar A;
2  int w,x,y;
3  w = A.op1(4,5);
4  y = w +1;
```

(b) w is used in a strict expression, block at wait for value.

Figure 8 Two uses of result variables. In this example, bar is a regular Mentat class.

overhead. A table approach with a unique mapping from a host identifier to its address would have been better.

A second disadvantage to using hostnames in MMPS names is that the naming scheme used is different on different platforms. On Unix systems a name is a *hostname* and a *port*, on the Intel using NX/2 a name is a *hostnumber* and a *pid*. This violates our objective of an invariant virtual machine interface. When we were supporting only two platforms this was not a problem; as we came to support more platforms it became difficult to manage.

3.2.2. Run-Time Dataflow Detection

The objective of the run-time dataflow detection is to dynamically detect and manage data dependencies between Mentat object invocations, mapping the resulting dependence graph onto a macro dataflow program graph. The data dependencies between Mentat object function invocations correspond to arcs in the MDF program graph.

The dataflow detection library routines monitor the use of certain variables (called *result variables*) at run-time to produce data dependence graphs. The basic idea is to monitor the use of Mentat objects, and the use of the results of Mentat object member function invocations. Informally, if at run-time we observe a variable `w` (Figure 8) being used on the left-hand side of a Mentat object member function invocation, we mark `w` as *delayed* and monitor all uses of `w`. Whenever `w` is delayed and `w` is used as an argument to a Mentat object member function invocation, we construct an arc from the invocation that generated `w` to the consumer of `w`. If `w` is not delayed, we use its value directly. Whenever `w` is used in a *strict* expression, we start the computation that computes `w`, and block waiting for the answer.

More formally, let `A` be a Mentat object with a member function

```
int 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(4,5);
```

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(4,5);.
```

The `mentat_object` member function `invoke_fn()` is called when a Mentat object member function is invoked. It creates a new computation instance for the computation, (i.e., a new program graph node is created), and marshals the arguments, both actual arguments (line 3, Figure 8(a)), and arguments that are computation instances (line 5, Figure 8(a)). If an argument is a computation instance, `invoke_fn`() adds an arc from the argument to the new computation instance it is constructing.

A *result variable* (RV) is a variable that occurs on the left-hand side of a Mentat assignment statement, e.g., `w` in Figure 8. It 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. An RV has an actual value if it has a value that may be used. To detect dataflow at run-time we monitor all uses of result variables, both on the left- and right-hand sides.

Each RV has a 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 the PRV set and never be a member of RVS. Membership in the PRV set is determined at compile time.

The run-time system performs run-time dataflow detection by maintaining a table of the addresses of the members of the RVS called the `RV_TABLE`. Each `RV_TABLE` entry contains the address of the RV, and a pointer to a computation instance. The computation instance corresponds to the last assignment to the variable. If the address of an RV is not in the `RV_TABLE`, then the RV is not in the RVS.

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

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

RV_DELETE((char*) rv_address);

force();

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

The function SET_ME() 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, it is overwritten. Thus, we are implementing the single assignment rule using the RV_TABLE and computation instances. SET_ME() is the mechanism for adding a PRV to RVS.

The function RV_DELETE() deletes the RV_TABLE entry associated with rv_address if one exists. Before the entry is deleted, its associated computation instance is *decoupled*. By decoupling the computation instance the compiler tells the RTS that this computation instance will never occur on the right-hand side of an expression again. This is the mechanism for removing a PRV from RVS.

The function force() is a local operation used to begin the execution of any program graphs that have been constructed so far in the object. It constructs the future lists from the program graphs consisting of computation instances and sends messages with the appropriate future lists to the appropriate objects.

The function RESOLVE() is called when the user program requires a value for a result variable. This is the case when a strict expression is encountered. If an entry in the RV_TABLE exists for rv_address, RESOLVE() calls force(), and blocks using a special predicate until the result is available. Once the result is available, RESOLVE() places the result into the memory to which the rv_address points.

Example 1: Simple Mentat object invocation. In Figure 8 two program fragments were presented to illustrate blocking versus non-blocking member function invocation. The MPL translations are shown in Figure 9 (a) and Figure 9(b) respectively. In Figure 9(c) the left-hand figure illustrates the program subgraph state before execution of the code fragment of (a). The actor labeled F executes the code fragment, creates the subgraph containing A, B, and C, and replaces itself with the subgraph. The result is the right-hand figure.

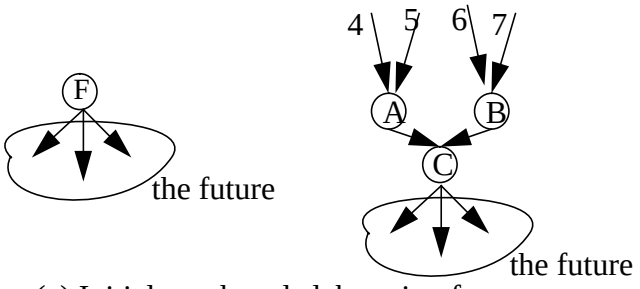
In Figure 9(d) there is no graph elaboration. Instead, because w is used in a strict expression, the compiler emits a RESOLVE to acquire the delayed value of w. The effect in this case is the execution of a short subgraph and a blocking RPC semantics.

```
bar A;
int w,x,y;
//w = A.op1(4,5);
// Add w to RVS, create a node in the subgraph, marshall arguments.
(*SET_ME((&w)))=A.invoke_fn(101,2,ICON_TO_ARG(4),ICON_TO_ARG(5));
//x = B.op1(6,7);
// Add xto RVS, create a node in the subgraph, marshall arguments.
(*SET_ME((&x)))=A.invoke_fn(101,2,ICON_TO_ARG(6),ICON_TO_ARG(7));
//y = C.op1(w,x);
// Add y to RVS, create a node in the subgraph, add arcs to subgraph
(*SET_ME((&y)))=C.invoke_fn(101,2,PRV_TO_ARG(&w,4,0),PRV_TO_ARG(&x,4,0));
// rtf(y);
// elaborate the current actor into the constructed subgraph
rtf(PRV_TO_ARG(&y,4));
```

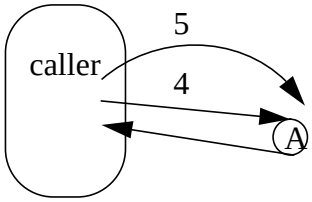
(a) Code transformation for (a)

```
bar A;
int w,x,y;
//w = A.op1(4,5);
// Add w to RVS, create a node in the subgraph, marshall arguments.
(*SET_ME((&w)))=A.invoke_fn(101,2,ICON_TO_ARG(4),ICON_TO_ARG(5));
//y = w +1;
y = (RESOLVE(&w),w) + 1;
```

(b) Code transformation for (b). Control flow blocks waiting for the result of the member function invocation, resulting in an RPC-like behavior.



(c) Initial graph and elaboration for fragment (a)



(d) Graph for (b).

Figure 9 Code transformations and generated graphs for code fragments of Figure 8. ICON_TO_ARG and PRV_TO_ARG are marshalling functions for integer constants and PRV's respectively. PRV_TO_ARG marshals the argument if the RV is actual. IF the RV is delayed, it constructs an arg_struct that points to the computation_instance that will generate the value.

This example illustrates the basic concepts used to detect data dependence at run-time. These particular code fragments were straight-line code. More complicated fragments, such as those that contain loops, conditionals, and multiple scopes, are handled in the same fashion.

3.2.3. Token Matching and Predicate Support

In the macro data flow model, as in pure data flow, tokens belonging to a particular computation must be matched [24][28]. For example, in Figure 10 (a) the tokens containing the values 4 and 5 must be

Figure 10 Token matching example. An actor may fire when there are tokens on each input arc.

matched. When both tokens are available, and they have been matched, we say the actor is enabled and may fire (execute). The matching process is complicated by the fact that there may be more than one instance of an actor in a particular program graph. For example, in Figure 10(b) there are three plus actors and four tokens, two for the first operand of plus, 5 and 3, and two for the second operand, 4 and 2. The problem is to correctly match the 5 and 4 to the $+_1$ execution and the 3 and 2 to the $+_2$ execution. The problem is further complicated by the fact that there may be multiple instances of $+_1$ executed, for example as a loop unrolls, or for multiple parallel invocations of a function.

The problem of identifying which tokens belong together has been solved in data flow using token coloring. Each actor invocation is assigned a unique color and then tokens destined for that actor instance are marked with that color. The problem then reduces to matching tokens of the same color. A similar coloring scheme is used in the Mentat RTS. Each computation is assigned a unique computation tag. The computation tag is formed from the MMPS name of the caller and an integer. Each invocation made by a particular caller is assigned a unique integer value used in forming the tag (we use a counter). Once the tokens are matched the actor may be scheduled for execution.

In Mentat the token matching problem has two components, token matching for bound persistent Mentat objects whose location is known, and token matching for unbound regular object member function invocations. The primary difference is that in the former case all of the tokens can be sent directly to the object, where matching the tokens can easily be done in the objects local address space. In the later case the location of the regular object is unknown -- in fact it does not yet exist, and will not exist until its tokens have been

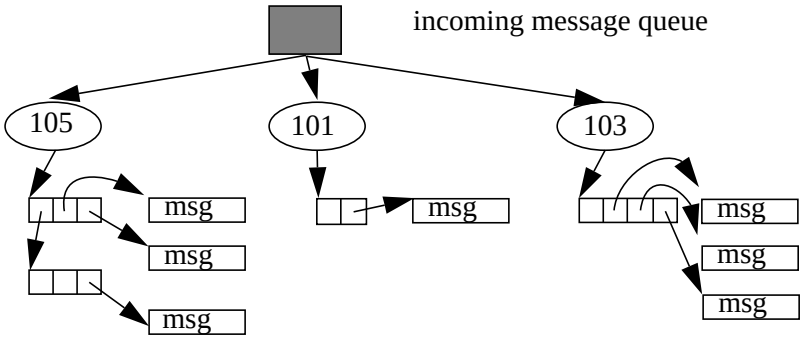


Figure 11 The predicate manager maintains its message database as a set of linked lists of work units. Incoming messages are forwarded to the correct list based on the function number.

matched. In this case the tokens must somehow be matched first by another agent, the token matching unit. The regular object token matching problem is complicated by the fact that the tokens are generated on different processors in a distributed memory system, requiring the use of a distributed algorithm. The realization of token matching for bound and unbound tokens is provided by the predicate manager and the token matching unit respectively.

3.2.4. Predicate Management

The predicate manager (PM) is a library linked into all Mentat applications and objects. The objectives of the PM are to support token matching, the MPL mselect/maccept statement, in particular the selective acceptance of function invocations, the evaluation of guards, and the enforcement of priorities.

The predicate management services are provided by the predicate manager and essentially allow the compiler to view the incoming message stream as a database against which predicates can be applied. For example, the compiler can construct a predicate that says that it is interested in messages for member functions 101, 103, and 105 [4], where 101, 103, and 105 take one, three and two arguments respectively. The predicate manager examines its database of received messages to determine if it has any complete 101, 103, or 105 *work units*. A work unit is a set of messages that all share the same computation tag, i.e., they correspond to the different arguments of the same instance of a member function invocation. This is shown in Figure 11. The collection of messages into work units corresponds to token matching in the MDF model.

[4] Member function names are mapped by the compiler to function numbers, thus an enq () operation on a queue might be mapped to function number 104.

The implementation is straightforward. For each function number (101, 103, etc.) the PM maintains a linked list of work units called an *actor list*. When a message arrives, the PM examines the destination field of the message to determine the function number, the argument number, and the computation tag. The PM scans the actor list of the specified function looking for a work unit with a matching computation tag. If a match is found, the message is added to the work unit and a check is made to determine if the work unit is complete, i.e., whether all of the required messages have arrived. If there is no match, a new work unit is created and added to the end of the list.

The predicate management interface to the compiler allows the compiler to specify the ordered list of functions in which the user is interested at any given point. Each function in the list corresponds to an accept in a mselect/maccept. The compiler generates code for a select/accept such that when the mselect/maccept is encountered at run-time, the functions' whose guards evaluate to true are in the ordered list, and so that they are in priority order.

At run-time when the mselect/maccept is encountered the list of functions is generated and a call to *block_predicate* is made. First, *block_predicate* calls *save_message*, which dequeues waiting messages from the message system and places them in appropriate work units. Then *block_predicate* traverses the ordered list of functions. For each function it finds the actor list for that function and scans the actor list for a complete work unit. If a complete work unit is found, the messages corresponding to the arguments are returned and the work unit deleted. At this point the compiler generated code invokes the appropriate object member function using the values contained in the messages as arguments. If no complete work unit is found then the next function in the ordered function list is similarly checked. This continues until a completed work unit is found or the list is exhausted. If none are found, *block_predicate* again calls *save_message*, this time specifying that *save_message* should block on message arrival if no messages are currently in the queue.

Example 2: The code fragment and transformation shown in Figure 12 illustrates how the predicate manager is used to support the mselect/maccept semantics.

3.2.5. Scheduling and Instantiation

The instantiation managers (IM's) perform four basic functions, Mentat object placement (scheduling), Mentat object instantiation, binding and name services, and general configuration and status information services. The first two services, scheduling and instantiation, are the most important and the most interesting. The last two, binding and status information services, are primarily bookkeeping, e.g.,

```
mselect {
            : maccept op1(int arg1,int arg2);
             break;
   (z>5)    : maccept op2(int arg1,int arg2);
            break;
};
```

(a) MPL mselect/maccept statement.

```
{
int pred_number;
mentat_message *msg1, *msg2;
predicate_manager pred=new predicate_manager(2);
pred->enable_operation(0, 101, 2);
if (z>5) pred->enable_operation(1, 102, 2);
pred_number = -1;
pred_number=pred->block_predicate(&msg1, &msg2);
 switch (predicate_number)
 {
 case 0 : {
   int arg1 = RESOLVE_MSG(int, msg1);
   int arg2 = RESOLVE_MSG(int, msg2);
   op1(arg1,arg2);
   delete msg1;
   delete msg2; }
   break;
 case1 : {
   int arg1 = RESOLVE_MSG(int, msg1);
   int arg2 = RESOLVE_MSG(int, msg2);
   op2(arg1,arg2);
   delete msg1;
   delete msg2; }
   break;
 }
delete pred;
}
```

(b) Translation of (a) generated by the MPL compiler.

Figure 12 Code transformations for run-time support of mselect/maccept. The code is transformed into calls to the predicate_manager class. Note the implementation of the guard.

reporting to the user the name of the objects running on a particular processor. Here we will confine our attention to scheduling and instantiation. A more detailed description can be found in [16].

The basic scheduling problem is to assign Mentat objects to processors in such a manner that total execution time of the application is minimized. The problem is complicated by three facts. First, nothing is known about the future resource requirements of the object being scheduled, or of the application as a whole. Second, the communication patterns of the application and the precedence relations between objects are unknown. Third, the current global state of the system is unknown, e.g., what are the utilizations and queue lengths of the processors. This third problem is further complicated by the fact that in distributed systems environments, there may be other users whose resource demands cannot be anticipated.

There is a rich literature on scheduling in distributed systems [8][10][19]. The general scheduling problem is NP-hard. Thus, much of the work in scheduling is based upon heuristics. Our scheduler, FALCON (Fully Automatic Load Coordinator for Networks) [16] is heuristic and based upon the work of Eager, Lazowska and Zahorjan [10] who developed a model for sender-initiated adaptive load sharing for homogeneous distributed systems. Their model uses system state information to describe the way in which the load in the system is distributed among its components.

FALCON falls into the classification schemes of [8] [19] as having the following characteristics:

- Distributed: The scheduling decision is distributed. Each decision is reached independently.
- Load Sharing: FALCON transparently distributes the workload by transferring new tasks from nodes that are heavily loaded to nodes that are lightly loaded.
- Adaptive: FALCON employs information on the current system state in making transfer decisions. Hence, it reacts to changes in the system state.
- Sender-Initiated: Congested nodes search for lightly loaded nodes.
- Static Assignment: Each task remains on the node to which it is assigned until the execution of the task and communication with other tasks is completed. In other words, there is no migration or reassignment after execution of a task is started.
- Stable: A task can only be transferred a fixed number of times between the nodes of the system. Transfers occur before the beginning of task execution while searching for a suitable node to handle the task. Thus, processor thrashing is avoided.

Scheduling decisions are reached using a distributed algorithm. Each node (or virtual node in the case of multi-computers) has an IM and a TMU, as shown in Figure 13. This view, taken by the IM's, is an extension of the simplified view of Figure 6, in which the distributed memory nature of the underlying hardware is recognized and accommodated.

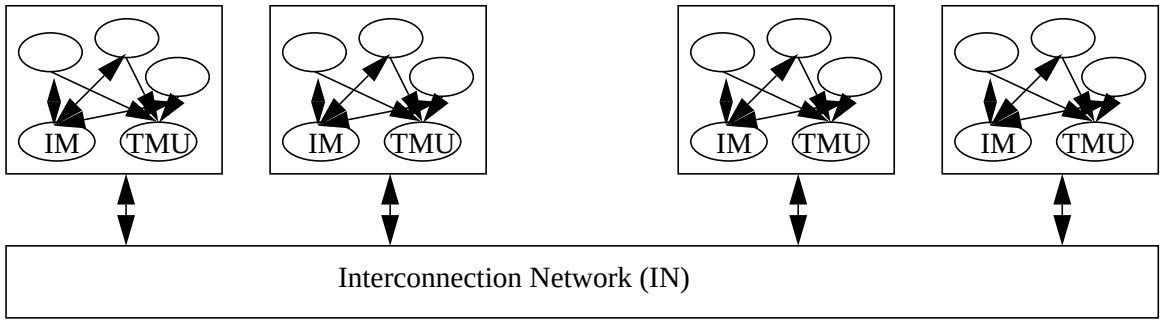


Figure 13 —IM and TMU view of the architecture. Hosts communicate via the interconnection network.

The scheduling decision on each IM consists of two sub-decisions: 1) determining whether to process a task locally or remotely (transfer policy), and 2) determining to which node a task selected for transfer should be sent (location policy).

The transfer policy that we have selected is a *threshold* policy: a distributed, adaptive policy in which each node of the system uses only local state information to make its decisions. No exchange of state information among the nodes is required in deciding whether to transfer a task. A task originating at a node is accepted for processing if the local state of the system is below some threshold. Otherwise, an attempt is made to transfer that task invocation request to another node. Note that only newly received tasks are eligible for transfer.

An issue in the implementation of a transfer policy is how the destination node should treat an arriving transferred task. One possibility is that it should treat it just as a task originating at the node: if the local state of the system is below threshold, the task is accepted for processing. Otherwise, it is transferred to some other node selected by the location policy algorithm. This can cause instability; the system may eventually enter a state in which the nodes are devoting all of their time to transferring tasks and none of their time to processing them.

Instability is overcome by the use of a control policy. The simple control policy that we use is to impose a static limit on the number of times that a task can be transferred. The destination node of the last allowed transfer must process the task regardless of its state. When the transfer limit is reached the IM must accept the task. The transfer limit depends on the number of available nodes and may be specified.

The location policy is invoked if the transfer policy does not accept the task for local instantiation or if a location hint specifies a different node for execution of the task. The three location policy algorithms that we have implemented are *random*, *round-robin* and *best-most-recently.*

We have found that scheduling policies such as those employed by FALCON are not in general appropriate for parallel computation systems. The reasons in retrospect are clear. The queuing theory results indicate that those schedulers will not perform well under heavy load - - but that is precisely where parallel programs operate. The programmer wants maximum utilization of available resources. The result is that two components of a data-parallel object may be scheduled on the same processor, doubling the execution time of the application. We have found that in order to make good scheduling decisions the scheduler must know whether an object is computationally *heavy* or *light*, and whether it is likely to be in the critical path. Then, if the object is heavy, and in the critical path, the scheduler should do whatever is required to find an idle node, even if that means visiting every node. On the other hand, much less effort can be expended scheduling light-weight, or non-critical path objects. Exploitation of this type of information has not been incorporated into the run-time system.

3.2.6. Token Matching

The token matching unit (TMU) is responsible for matching tokens for regular object member function invocations. The basic problem is similar to that of token matching for bound objects discussed earlier. The difference is that there is no bound object to which to send the tokens where matching can occur, since the object that will perform the member function does not yet exist.

Token matching is accomplished using a distributed algorithm. There is one token matching unit for each node (or virtual node) in the system. The number of TMUs grows with system size so that the token matching does not become a bottleneck. The algorithm is both simple and scalable,[5] and has four stages that are executed for each regular Mentat object member function invocation, unbound token routing, token matching, instance acquisition, and token delivery.

During unbound token routing, tokens which are generated at Mentat objects and are destined for an unbound regular object member function invocation are routed to the appropriate TMU. The appropriate TMU is determined using a hash function on the computation tag of the member function invocation instance. This is similar to the way hashing is done on token colors in dynamic dataflow[9][24]. By using the same hash function everywhere, we guarantee that all of the tokens for a particular invocation arrive at the same TMU. The hash functions are very simple and uniformly distribute the tokens to the TMU's. A result of using a hash function to route tokens is that each token typically makes one hop in the first stage of the algorithm.

Once the tokens have arrived at the designated TMU they are stored in a token database. The database is organized in a manner similar to that used in the predicate manager. Tokens are collected into unmatched work units. Because tokens may both be large and unmatched for a long time, tokens may be stored on disk when the memory allocation for the TMU is exhausted.

When all of the tokens for a particular computation tag have arrived, the TMU issues a *create* request to the local IM. We call this *instance acquisition*. The TMU does not block on the create request. To do so would reduce parallelism. Instead, it continues to receive unmatched tokens, and issues additional instantiation requests. Thus, several instantiation requests may be outstanding at any given time.

The final phase, token delivery, begins once the IM has replied to the instantiation request. The IM returns a bound Mentat object name. The named object has been selected by the IM to service the regular object member function. The TMU extracts from the token database the tokens for the computation and

---

5. An earlier method described in [17] was not scalable, it has been replaced.

forwards them to the named object. Thus, each token is transported two hops, once to the TMU, and once from the TMU to the object. When the tokens arrive at the bound object, they are re-matched using the predicate manager as described earlier, and the member function is executed to completion.

4. Implementation

To date Mentat has been implemented on seven platforms. The implementations fall into two categories, Unix-based workstation networks, the Sun 3, Sun 4, Silicon Graphics, and IBM RS 6000; and distributed memory MIMD multicomputers, the TMC CM-5, Intel iPSC/2, and iPSC/860 Gamma. Within each category the implementations are similar, although there are some significant differences between the multicomputers. Rather than describe each implementation in detail we present the core features of the Unix and multicomputer implementations. We then present the performance of the run-time system primitives on two well known and representative platforms, the Sun SparcStation 2, and the Intel iPSC/860.

4.1. Unix Workstations

The multitasking Unix implementation of the run-time system is by far the most complex of our implementations, primarily because it requires the most system-specific code; the operating system communication support is very weak, interrupts of various forms must be managed, and the processes model, while offering many options, differs in subtle ways from platform to platform.

Mentat objects are implemented by Unix processes that are fork/exec'd by the instantiation manager. Multiple Mentat objects may reside on each Unix host and may execute concurrently. All communication between objects, including intra-host communication, is implemented in MMPS using UDP packets and an interrupt-driven, stop-and-wait protocol.

Two different scheduling architectures are supported in the Unix implementation. The first, shown earlier in Figure 13, is a degenerate form of the second. In the first architecture, all processors are logically equivalent and equidistant. In the second, general form, hosts may be partitioned into *families* and *clusters* (Figure 14). A family of hosts all share the same attributes, such as host type, number of processors, physical memory available, clock speed, MIPS rating, MFLOP rating, TMU memory, whether kernel memory may be used to acquire load information, scratch and binary paths, and so on.

While families represent processor attributes, clusters indicate locality information that is used in the location policy during object scheduling. Members of a cluster are presumed by the instantiation managers to be closer together, e.g., they may reside on the same network segment. The instantiation manager exploits this information and tries to schedule new objects on the same cluster as the creator (unless the creator has specified

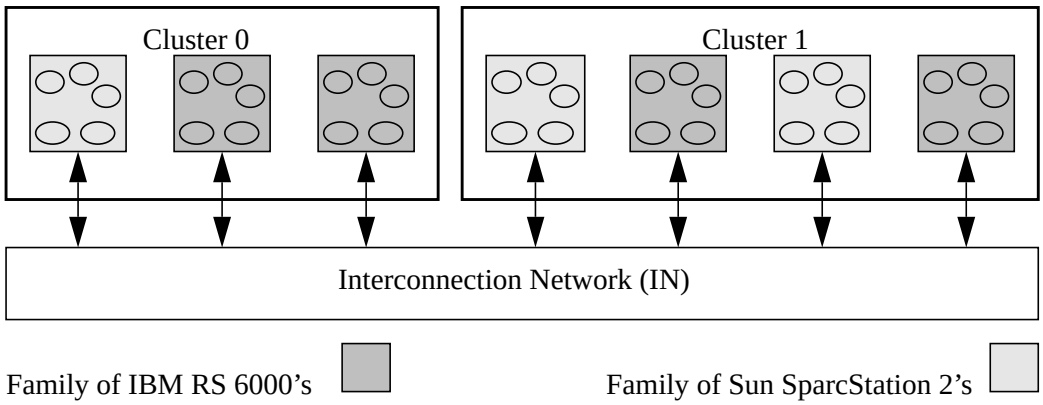


Figure 14—Families and clusters. Families of nodes share attributes. Clusters indicate locality for scheduling.

via a location hint that the object is heavy weight, and may be safely placed far away). Only if an idle node cannot be found in the local cluster is the object transferred to another cluster.

Currently different host capabilities are not exploited by the scheduler. Instead, the hosts are treated uniformly for scheduling purposes. Also, we are currently limited to either homogeneous sets of workstations, or to workstations that have the same data representation and alignment characteristics, e.g., Sun 4's, IBM RS 6000's, and Silicon Graphics.

Finally, unlike multicomputers such as the iPSC/2 and iPSC/860, networks of workstations typically have other users. These users present several challenges, they introduce extraneous processor loads which the scheduler must detect and avoid, they introduce network traffic that slows down application performance, and most importantly, they complain that interactive application performance is negatively impacted. This last point deserves some elaboration. What we have found is that lowering a process's priority is not sufficient to keep it from disturbing interactive users. This is true because priority only affects the CPU usage. Mentat application memory use and overall network traffic induced by the Mentat application are what interactive users notice. Mentat applications may page interactive users out, forcing multiple page faults when the user moves the mouse. Induced network traffic makes file server access much slower, similarly affecting interactive application performance.

We have addressed this problem in three ways. First, Mentat may be *suspended* on a particular host. This has the effect of preventing any further jobs from being scheduled on that host, and of suspending all Mentat tasks on that host. Suspension of Mentat tasks may have the side-effect of blocking an entire application, and therefore should be used with caution. Second, the instantiation managers can be set to periodically check and see if there is any activity at the console on the host. If there is, the instantiation manager suspends its'

children and refuses further scheduling requests until the console is idle for a specified period. There is naturally a trade-off between the responsiveness to interactive activity (determined by the period), and the overhead of checking for that activity. Finally, Mentat can be "niced" to a low level of priority. While, as noted earlier, this does not have much affect on performance, it does impact user perception. When users check the active processes on their workstation they see that the Mentat processes have a lower priority than theirs. They then assume that something else is responsible.

4.2. Multicomputers

The three multicomputers to which we have ported Mentat differ in several important respects - yet they are very similar in terms of their communication support; all three provide for asynchronous, guaranteed delivery of arbitrary size messages. The differences lie in their level of process support. They fall into a spectrum from no process support on the TMC CM-5, to almost Unix-like process support on the iPSC/2.

On the CM-5 there is one user processes per processor and all processors must execute the same executable image[6]. Thus, there is no support for dynamically loading different Mentat object executables on different processors, let alone multiple Mentat objects per processor. Instead all Mentat objects, the instantiation manager, the token matching unit, and the main program, must be linked into one large executable. A switch statement is then used to select which object to execute.

At the other extreme is the Intel iPSC/2 which has a truly multitasking operating system. The iPSC/2 supports up to twenty processes per processor, and the capability to load new executables. Thus, the iPSC/2 execution model is very similar to the Unix model.

In the middle is the Intel iPSC/860 Gamma. The Gamma supports exactly one process per processor. However, the executable image on the processor may be changed by issuing a load command to the operating system. Since both the Gamma and the CM-5 allow only one process per processor, a different process mapping paradigm is adopted.

If each Gamma node ran a copy of the instantiation manager, then no user processes could be scheduled on the Gamma. Instead, the allocated Gamma sub-cube is partitioned into sets of virtual nodes comprised of a number of processors. Each virtual node has a single instantiation manager responsible for scheduling within the virtual node, and a token matching unit (Figure 15). All virtual nodes contain the same number of processors and the user may select the number of processors at configuration time. The first two processors of each virtual node are reserved for the instantiation manager and token matching unit.

6. The CM-5 is actually multitasking, but a single user application is restricted to a single task per node.

Therefore, the minimum virtual node size is four. Typically there are at least sixteen processors in a virtual node. When an object instantiation request is accepted within a virtual node, the instantiation manager executes a *load* onto one of the virtual node processors. If the request is not accepted within the virtual node, the request is passed on to the instantiation manager of another virtual node.

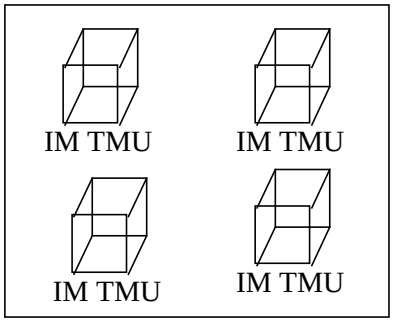


Figure 15 —Virtual nodes on the Intel iPSC/860 Gamma. A 32 processor cube has been partitioned into four virtual nodes of eight processors each. Links connecting the virtual nodes are not shown.

4.3. Performance

Overhead is the friction of parallel processing. The performance of Mentat applications depends upon the overhead of the Mentat run-time system calls needed for program execution. Good performance requires that the run-time overhead be kept as low as possible. The performance of each Mentat run-time system component is presented for a network of Sun SparcStation 2 workstations and the Intel Gamma iPSC/860. Two types of run-time system overhead are measured: library calls and service requests. A library call is a function that is performed by the run-time system on behalf of the user application. Most library calls are simple function calls that operate in the local address space of the caller without any communication. A service request is a special type of library call that requires participation of a Mentat daemon process and requires communication. All measurements were taken when the machine or network usage was as low as possible.

The functions timed are those that are most critical for good performance, or in the case of the null RPC, that capture all of the overhead terms. The functions we have timed are:

- block_predicate - We measure the time required to search the message database. All of the necessary messages have already arrived. Thus, there is no blocking waiting for a message. Two predicates were enabled, the match was found for the first predicate. The cost includes the token matching.
- new and delete a computation instance - Allocate a computation instance, initialize its members, and destroy it.
- add an arc - Add an arc between two computation instances.

- construct a future - Construct a future from the computation instances. Two cases are given, for a short RPC-like future, and for an unrolled loop with one hundred actor invocations.
- RV_TABLE insert - We measure the time required to insert an entry into the RV_TABLE. Because the table is implemented by a hash table, the insertion cost is constant.
- RV_TABLE look up - Look up an entry in the RV_TABLE.
- new and delete a mentat_message - Allocate a mentat_message, initialize its members, and delete it.
- message transport - Send a one byte message. This cost includes in the Unix implementation over three hundred bytes of header. The time is measured by starting a timer, sending the message from A to B, on receipt B sends a message back to A, on receipt A stops the timer.
- message send - Send a one byte message but don't wait for delivery. This is measured by starting a timer, sending from A to B using the asynchronous send capability, and stopping the timer. The message has been delivered to the communication system (MMPS) only. MMPS will asynchronously transport the message.
- Null RPC - Perform a blocking RPC that takes an integer parameter and returns an integer. No computation is performed. The time taken is pure overhead.
- Total Mentat overhead - Null RPC time - 2*(message transport time). This captures Mentat overhead versus hand-written send/receive.
- Object instantiation - The time required to create a new Mentat object. This includes all scheduling time. Because measurements were done on an idle system these results should be viewed as lower bounds, not expected values.
- Regular object instantiation latency - Here we measured the time for a single blocking RPC call on a regular mentat object. This includes the time to construct the graph, transport the tokens to the TMU, match the tokens at the TMU, have the TMU acquire an instance (via the IM), transport the tokens from the TMU to the object, match the tokens at the object, and return the result to the caller. Two different times are given, with and without *reuse*. Regular object reuse occurs when an IM reuses an existing object, i.e., it does not load a new instance. Instead it just returns the name of the existing object. This saves the time required to perform the load, or the fork/exec.

Sun SparcStation 2

The Sun configuration consisted of a collection of 8 Sparc2s connected by ethernet. Each Sparc processor runs at 40 MHz and was configured with 32 MBytes of real memory. The run-time system

performance is presented in Table 1. Because the communication time variance is so large all measurements

Library Functions	**Cost uSec**
block predicate (w/o delay)	575
new and delete computation instance	71
add arc	4
construct a future - short	55
construct a future - 100 actors	2300
RV_TABLE insert	14
RV_TABLE lookup	10
new and delete a message	36
message transport	2,000[a]
message send	480
null RPC	6,000[a]
Total Mentat overhead	2,000[a]
Run-time Services	
object instantiation/deletion	140,000/6,000[a]
regular object instantiation latency	150,000[a]
regular object latency with reuse	20,000[a]

a. Rounded to the nearest millisecond.

Table 1. Performance of Mentat RTS on Sun SparcStation 2 Network.

involving the communication system have been rounded to the nearest millisecond.

Intel iPSC/860 Gamma

The Gamma performance data were collected on the 64-node Intel Gamma iPSC/860 at Caltech using an 8 node partition. Each i860 processor runs at 40 MHz and contains 16 MBytes of real memory. The Gamma nodes do not have virtual memory. The run-time system performance is shown in Table 2.

4.4. Performance Observations

On the Sun 4's cost of communication clearly dominates all other overhead terms. This is no surprise, given the UDP/IP implementation. The cost of dynamic graph construction, monitoring RV's, creating

Library Functions	**Cost uSec**
block predicate (w/o delay)	570
new and delete computation instance	31
add arc	3
construct a future - short	15
construct a future - 100 actors	1300
RV_TABLE insert	13
RV_TABLE lookup	9
new and delete a message	13
message transport	180
message send	14
null RPC	2,000
Total Mentat overhead	1,600
Run-time Services	
object instantiation	500,000/250
regular object instantiation latency	503,000
regular object latency with reuse	Not available

Table 2. Performance of Mentat RTS on the Intel iPSC/860 Gamma.

computation instances, adding arcs, and constructing future lists, is quite small (190 uS) when compared to the message transport costs. This is good news, and justifies the use of dynamic graphs[11]. That we have not captured all of the overhead terms is also clear - the remainder is in the two block predicate calls (1150 microseconds) and in numerous small ancillary functions. Finally, regular object reuse results in a large time savings, 20 mS versus 150 mSec.

On the Intel iPSC/860 Gamma, communication does not dominate to nearly the same degree. Communication is an order of magnitude faster than on the Sun 4. Once again this is not a surprise, as the Gamma has an optimized communication system and special communication hardware. The result is that the RTS software overhead, which remains fairly constant between these two platforms, dominates on the Gamma. Object instantiation is also very slow. This is the result of a very slow implementation of the NX “load” call on the Gamma.

5. Future Work and Summary

The application of the object-oriented programming paradigm to parallel computing depends on the efficient implementation of a supporting parallel run-time system. The Mentat run-time system provides a portable run-time environment for the execution of parallel object-oriented programs. The run-time system supports parallel object-oriented computing via a virtual macro dataflow machine that provides services for object instantiation and scheduling, select/accept management, remote member function invocation, and object scheduling and instantiation. The run-time system achieves portability by using a layered virtual machine model that hides platform-specific details from the user and the compiler. As of this writing the run-time system has been implemented on platforms that include networks of Sun 3's, Sun 4's, the IBM RS 6000, and Silicon Graphics workstations, as well as multicomputers such as the Intel iPSC/2 and iPSC/860.

We presented the overhead costs for the dominant run-time services on two platforms, the Sun SParcStation 2 and the Intel iPSC/860. Performance on applications is very good [12][14][15] and leads us to the conclusion that the run-time system overhead is tolerable.

Future work on the Mentat run-time system falls into three categories, porting to additional platforms, generating a threads-based implementation, and providing heterogeneous metasystem support. The quest to port to an expanding list of platforms continues. We have set our sights in the near term on two additional workstation families, the DEC alpha and the HP PARisc. In the multicomputer domain we are targeting the Intel Paragon, the TMC CM-5, and the Cray MPP. Toward this end we are further isolating machine, operating system, and compiler dependencies so that porting will become even easier.

In a thread-based implementation of the run-time system Mentat, objects will no longer necessarily be physically address space disjoint. Communication between Mentat objects within the same address space will use messages implemented using shared memory. There are several advantages to a thread-based implementation, intra-host communication is much faster, shared memory multiprocessor (e.g., KSR or multiprocessor Sparc) implementations will be much faster than is possible with disjoint address spaces, and limitations on Mentat applications caused by underlying unitasking operating systems (e.g., iPSC/860 and CM-5) will be eliminated. A significant disadvantage is that many user-defined library routines are not re-entrant. This can cause timing-dependent bugs that do not exist in the current implementation.

A metasystem is a collection of heterogeneous hosts, scalar workstations, vector processors, SIMD machines, and shared and distributed memory MIMD machines connected by a multilayer, heterogeneous interconnection network. Our objective is to integrate these hosts into a system that provides the illusion of one large virtual machine to users. As part of the Mentat metasystem testbed project [13] we are extending the

Mentat run-time system into a heterogeneous environment. Issues that must be addressed include data alignment, data coercion, and scheduling, in particular, automatic problem decomposition and placement [27].

Acknowledgments

This research was performed in part using the Intel iPSC/860 Gamma operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by JPL.This research was also performed in part using the TMC CM-5 owned by the Northeast Parallel Architectures Center (NPAC) located at Syracuse University.

6. References

[1] T. Agerwala and Arvind, "Data Flow Systems," *IEEE Computer*, vol. 15, no. 2, pp. 10-13, February, 1982.

[2] R. F. Babb, "Parallel Processing with Large-Grain Data Flow Techniques," *IEEE Computer*, pp. 55-61, July, 1984.

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

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

[5] A. Beguelin et al., "HeNCE: Graphical Development Tools for Network-Based Concurrent Computing," *Proceedings SHPCC-92*, pp. 129-136, Williamsburg, VA, May, 1992.

[6] F. Bodin, et. al., "Distributed pC++: Basic Ideas for an Object Parallel Language," *Proceedings Object-Oriented Numerics Conference*, pp. 1-24,Sunriver, Oregon, April 25-27, 1993.

[7] J. C. Browne, T. Lee, and J. Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment," IEEE Transactions on Software Engineering, pp. 111-120, vol. 16, no. 2, Feb., 1990.

[8] T. L. Casavant, and J. G. Kuhl, 'A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems,' *IEEE Transactions on Software Engineering*, vol. 14, pp. 141-154, February, 1988.

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

[10] 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.

[11] A. S. Grimshaw, "The Mentat Computation Model - Data-Driven Support for Dynamic Object-Oriented Parallel Processing," Computer Science Technical Report, CS-93-30, University of Virginia, May, 1993.

[12] A. S. Grimshaw, E. A. West, and W.R. Pearson, "No Pain and Gain! - Experiences with Mentat on Biological Application," *Concurrency: Practice & Experience*, pp. 309-328, Vol. 5, issue 4, July, 1993.

[13] A.S. Grimshaw, J.B.Weissman, E.A. West, and E. Loyot, "Meta Systems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems," TR-92-43, Department of Computer Science, University of Virginia, December, 1992.

[14] A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, pp. 39-51, May, 1993.

[15] A. S. Grimshaw, W. T. Strayer, and P. Narayan, "Dynamic Object-Oriented Parallel Processing," *IEEE Parallel & Distributed Technology: Systems & Applications*, pp. 33-47, May, 1993.

[16] A. S. Grimshaw and V. E. Vivas, "FALCON: A Distributed Scheduler for MIMD Architectures", *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 149-163, Atlanta, GA, March, 1991.

[17] A. S. Grimshaw, "The Mentat Run-Time System: Support for Medium Grain Parallel Computation," *Pro-*

ceedings of the Fifth Distributed Memory Computing Conference, pp. 1064-1073, Charleston, SC., April 9-12, 1990.

[18] 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.

[19] A. Hac, "Load Balancing in Distributed Systems: A Summary", *Performance Evaluation Review*, ACM, vol. 16, pp. 17-25, February 1989.

[20] 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.

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

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

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

[24] V. P. Srini, "An Architectural Comparison of Dataflow Systems," *IEEE Computer*, pp. 68-88, March, 1986.

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

[26] J. B. Weissman, A. S. Grimshaw, and R. Ferraro, "Parallel Object-Oriented Computation Applied to a Finite Element Problem," to appear in *Scientific Computing*, 1993.

[27] J. B. Weissman and A. S. Grimshaw, "Multigranular Scheduling of Data Parallel Programs," Computer Science Technical Report, CS-93-38, University of Virginia, July, 1993.

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