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

ANDREW S. GRIMSHAW, JON B. WEISSMAN, and
W. TIMOTHY STRAYER
University of Virginia

Mentat is an object-oriented parallel processing system designed to simplify the task of writing portable parallel programs for parallel machines and workstation networks. The Mentat compiler and run-time system work together to automatically manage the communication and synchronization between objects. The run-time system marshalls member function arguments, schedules objects on processors, and dynamically constructs and executes large-grain data dependence graphs. In this article we present the Mentat run-time system. We focus on three aspects—the software architecture, including the interface to the compiler and the structure and interaction of the principle components of the run-time system; the run-time overhead on a component-by-component basis for two platforms, a Sun SparcStation 2 and an Intel Paragon; and an analysis of the minimum granularity required for application programs to overcome the run-time overhead.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; *objected-oriented languages*; D.3.4 [**Programming Languages**]: Processors—*run-time environment*

General Terms: Languages, Performance

Additional Key Words and Phrases: Dataflow, distributed memory, MIMD, 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 software for

This work was partially funded by NSF grants ASC-9201822 and CDA-8922545-01, and NASA NGT-50970. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.
Authors' address: University of Virginia, Thornton Hall, Charlottesville, VA 22903-2442; email: {grimshaw; weissman; strayer}@virginia.edu.

parallel machines as well [Beck 1990; Bershad et al. 1988; Bodin et al. 1993; Grimshaw 1993a; Lee and Gannon 1991; Smith et al. 1989]. Software for parallel machines must provide efficient run-time support if it is to achieve acceptable performance. This is particularly true for distributed-memory MIMD machines where communication and synchronization costs can be large. 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, efficient run-time support must be provided.

Mentat is an object-oriented parallel processing system designed to simplify the task of writing portable, parallel applications software [Grimshaw 1993a; Grimshaw et al. 1993a; 1993b; Weissman et al. 1994]. The fundamental objectives of Mentat are to (1) provide easy-to-use parallelism, (2) facilitate the portability of applications across a wide range of platforms, and (3) achieve good 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 the run-time costs incurred, is the topic of this discussion.

Mentat has two primary components: the Mentat programming language (MPL) and the Mentat run-time system (RTS). 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 of sufficient computational weight to allow efficient parallel execution. MPL programs manipulate instances of these special classes by invoking member functions on them just as if they were instances of 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.

Mentat differs from other systems in several ways.[1] First, Mentat is the only system in the literature that combines the object-oriented paradigm with coarse-grain dataflow. Further, Mentat dynamically detects and manages data dependencies as they develop, rather than at compile-time. This facilitates larger-grain computation by eliminating many small-grain control actors which are inefficient in a distributed-memory environment. Second, Mentat supports both task and data-parallelism, not just data-parallelism as in Bodin et al. [1993], Cheung and Reeves [1992], Fox et al.

[1]A more complete discussion of related work appears in Section 5.

[1990], Hatcher et al. [1991], and Loveman [1993]. Third, Mentat operates over a spectrum of architectures, from loosely coupled heterogeneous networks of workstations to tightly coupled multicomputers. Finally, Mentat has a scalable, distributed, control mechanism. This includes both the mechanism used to construct and modify program graphs at run-time as well as the scheduling mechanism. We must point out, however, that Mentat is not suitable for all applications: Mentat is useful for medium-to-coarse-grain applications. If the computation granularity of an application is too small, performance will suffer.

Performance aspects of Mentat are presented in Grimshaw et al. [1993a; 1993b]. For a general overview of Mentat and of the Mentat parallel processing philosophy see Grimshaw [1993b]. In this article we present the Mentat run-time system which supports the Mentat programming language. Our objectives are threefold: first, to describe the inner workings of the run-time system, its software architecture, and the structure and interaction of its components; second, to present the run-time costs incurred by Mentat applications on two platforms; and third, to provide insight into the minimum computation granularity required on those platforms. We begin with background information on the macro dataflow model of computation and the Mentat programming language. Macro dataflow 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 a network of Sun SparcStation 2 workstations and an Intel Paragon.

2. BACKGROUND

2.1 Macro Dataflow

The macro dataflow (MDF [Grimshaw 1993a]) model is a medium-grain, data-driven computation model inspired by dataflow [Agerwala and Arvind 1982; Dennis 1975; Srini 1986; Veen 1986]. 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. An actor is a function that maps inputs to outputs. 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, have arrived.

Macro dataflow differs from traditional dataflow in three ways. First, the computation granularity is larger than in traditional dataflow [Babb 1984; Beguelin et al. 1992; Browne et al. 1990]. Actors are high-level functions such as matrix-multiply specified in a high-level language, not primitive operations such as addition. The required granularity in the Mentat MDF model varies from platform to platform, but is in the thousands of instruc-

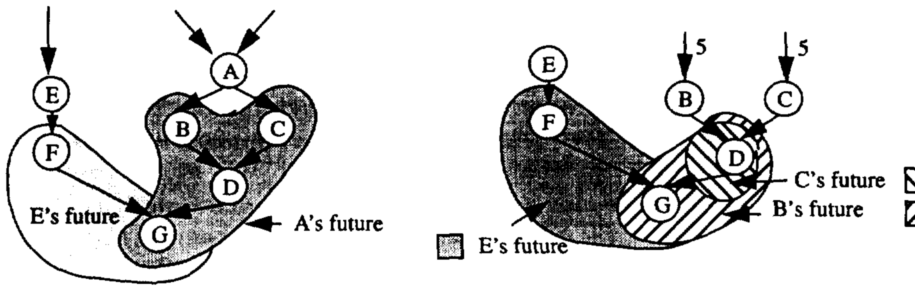


Fig. 1. (a) A macro dataflow subgraph. The future of an actor A is shown; (b) new futures after actor A from (a) returns a value.

tions rather than tens of instructions. In Section 4.4, we present a detailed analysis of the required granularity. 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. Finally, program graphs are not fixed at compile-time; instead, program graphs are constructed at run-time by observing the data dependencies as execution unfolds.

Program graphs are represented 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. In this respect MDF futures are similar to a parallel form of a continuation [Abelson et al. 1985]. For example, consider the program graph fragment of Figure 1(a). A's future is shown by the shaded area enclosing the actors (B, C, D, G). The future of A 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.

Within the MDF execution model, actors such as A may execute when their tokens arrive. A's future is passed along with the tokens. 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 1(b). B and C are now enabled and may execute, since they have a token (value) on each input arc.

[2]MDF futures should not be confused with Multilisp futures [Halstead 1985]. Multilisp futures represent a promise to deliver a value in the future.

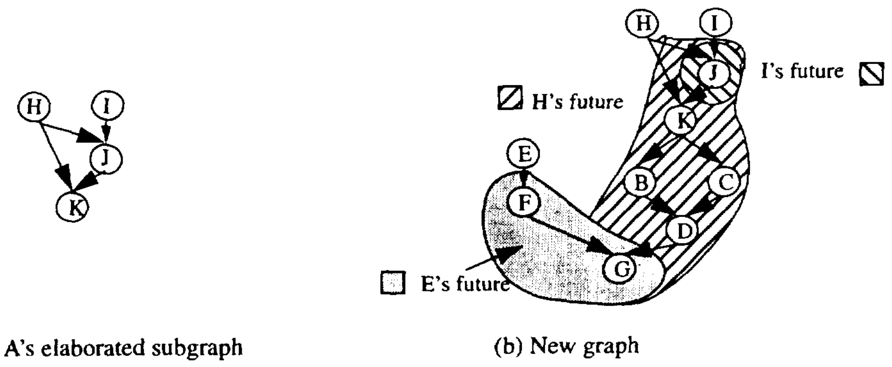


Fig. 2. Actor elaboration. The actor *A* from Figure 1 may elaborate into the subgraph of (a). The resulting graph is shown in (b). This figure also illustrates an optimization with respect to futures. Although *H* and *I* have the same future, a single copy of this future is sent from *H* to *J*, resulting in a smaller message.

Alternatively, *A* may elaborate itself into an arbitrary subgraph. Subgraph elaboration takes place when an actor's implementation is realized using other actors connected together to form a small subgraph. Which actors, and how they are connected together, is a language issue and will be discussed later. For now, suppose that *A* elaborates into the subgraph shown in Figure 2(a). The new state is shown in Figure 2(b). In this case the graph has grown, rather than contracted as in Figure 1(b). 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.

2.2 The 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 information to the compiler and run-time system.

The most important extension to C++ is the keyword "mentat" as a prefix to class definitions, as shown in Figure 3. This keyword indicates to the compiler that the member functions of the class are of sufficient computational weight to be worth executing concurrently. 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 class to service each invocation of a member function from that class, even while other instances of the same class already exist.

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

```
mentat class bar {
// private member functions and variables
public:
  int op1(int,int) ;
  int op2(int, int) ;
} ;
```

Fig. 3. A Mentat class definition. Without the keyword "mentat," it is a legitimate C++ class definition.

Instances of Mentat classes are called *Mentat objects*. Each Mentat object possesses a unique name, an address space, and a single thread of control. Mentat objects are *logically* address-space disjoint. The current implementation of Mentat objects is address-space disjoint, although a thread-based implementation is planned. Because Mentat objects are address-space disjoint, all communication is via member function invocation and is accomplished by RPC. 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. The single thread and disjoint address space provide constrained access to contained variables and state manipulated by the object, and they prevent race conditions.

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 necessarily an error. If the variable names a regular Mentat class, the underlying system instantiates a new Mentat object to service the member function invocation. On the other hand, instances of persistent Mentat classes must be bound explicitly through other language mechanisms not discussed here.

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 nonblocking, 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 nonblocking 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, 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 are convenient for implementing class libraries (e.g., 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.

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. 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 2(b).

2.2.3 *The mselect/maccept Statement.* The MPL mselect/maccept 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. Unlike Ada, rendezvous semantics are not supported, and the caller does not block. The mselect/maccept statement gives the programmer the ability to control access to the object by specifying conditions for access. It also permits the construction of more elaborate and explicit synchronization constructs than are implicit in the language.

2.2.4 *Task and Data Parallelism in Mentat.* Task parallelism is realized using the language features just described. Instances of Mentat classes are created by users and used much as their C++ counterparts are. For each instance a thread and a logically disjoint address space are created. The compiler generates code to marshall arguments and to construct macro dataflow graphs at run-time. The nodes in the graphs are member function invocations; the arcs model the formal parameters; and tokens represent the actual parameters.

Data parallelism in Mentat applications is currently realized manually using the task parallelism features and encapsulation. The basic idea is simple. A data-parallel object is implemented using a persistent Mentat class that encapsulates several persistent Mentat objects. Member functions of the class are implemented by calling the same member function on the contained persistent Mentat objects which contain the data for the class. For example, a Mentat matrix class has been implemented by decomposing the matrix into k submatrices [MacCallum and Grimshaw 1994]. Each submatrix is implemented using a persistent Mentat object. Member functions that manipulate the whole matrix are treated as task-parallel invocations by the caller. Internally they are implemented by appropriately manipulating the submatrices. This is known as intraobject parallelism. The result is a program graph elaboration as described earlier in which the program subgraph has a node for each contained Mentat object. This results in a combined task- and data-parallel program.

Similar techniques can be used for a wide range of data structures, not just regular, dense, two-dimensional arrays. The difficulty with this method for realizing data parallelism is that, while it is straightforward for the programmer, it is somewhat tedious. The programmer is responsible for managing the iteration space, data domain decomposition, and the exchange of boundary regions for some problems. The iteration must be performed over the contained Mentat objects and their contained data elements.

To eliminate the need to manually generate this rather tedious code we are exploring the incorporation of data-parallel language features into the Mentat programming language [West and Grimshaw 1995]. The language extensions are similar to those being developed by Lee and Gannon [1991], Hatcher et al. [1991], Larus et al. [1992], and others. Our work is significantly different in that the result will be a language that combines task and data parallelism in a seamless fashion, while the above languages are data parallel only.

3. THE RUN-TIME SYSTEM (RTS)

MPL programs are executed on a virtual macro dataflow machine implemented by the RTS. 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. Token matching is the process of matching tokens (messages) that are the actual arguments of the same Mentat object-member function invocation. The RTS constructs program graphs, manages communication and synchronization, performs token matching, performs object instantiation and scheduling, and allows selective message reception to support an ADA-like select/accept semantics. The virtual machine 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/860 and Intel Paragon, both using NX/2 [Intel 1988], the IBM SP-2, and networks of Sun, Hewlett-

Packard, Silicon Graphics, and IBM RS/6000 workstations using UDP packets and Unix sockets.

3.1 The Virtual Machine

The virtual macro dataflow machine 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 token matching for persistent Mentat objects. Object scheduling and instantiation are provided by instantiation managers (IMs), and unbound token matching by the token-matching units (TMUs). The IMs and TMUs 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 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 [Grimshaw et al. 1990]), 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.

Each Mentat object and daemon have a unique name, address space, and thread of control. Communication between objects is solely through the message-passing system—there is no shared memory.

3.2 System Services

There are five basic services provided by the run-time system: (1) object naming and basic communication, (2) dataflow detection, (3) token matching for persistent objects, (4) token matching for regular objects, and (5) object scheduling and instantiation. As discussed earlier, the first three are implemented by libraries and the latter two 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 computation_instance, mentat_object, mentat_message, and predicate_manager.

Class computation instance. The RTS keeps track of Mentat object-member function invocations at run-time using *computation instances*, which correspond to nodes in an MDF program graph. They contain 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

```
class metat_object {
    object_name i_name ;
public:
    CIP invoke_fn(int fn_number,int arg_count,arg_struct ...) ;
    // invoke_fn marshals the arguments. fn_number indicates the function,
    // arg_count the number of arguments
    void create( ) ; // instantiate new back-end
    void destroy( ) ; // destroy the back-end
} ;
```

Fig. 4. Partial interface of the front-end mentat_object class.

acquire the value that is the result of the operation. A CIP is a computation instance pointer.

Class 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. Mentat class-member function names are mapped by the compiler to function numbers; thus a pop( ) operation on a queue might be mapped to function number 104. Member function invocation involves using the front-end as a surrogate for the back-end server object. The front-end mentat_objects are essentially object names and a set of member functions used to communicate with the back-end server. The compiler generates the server loops that implement the back-ends and code to manipulate mentat_objects. Three member functions defined on every mentat_object, invoke_fn( ), create( ), and destroy( ) are shown in Figure 4.

The member function invoke_fn( ) is called when a Mentat object-member function is invoked. For example, to invoke A.pop( ) the compiler would emit a A.invoke_fn(104,0) to call function 104 on object A with no arguments. Invoke_fn( ) creates a new computation instance for the computation, (i.e., a new program graph node is created) and marshalls the arguments, both actual arguments (line 3, Figure 5(a)) and arguments that are computation instances (line 5, Figure 5(a)). If an argument is a computation instance, invoke_fn( ) adds an arc from the argument to the new computation instance it is constructing. The functions create( ) and destroy( ) are called when a persistent Mentat object is instantiated and destroyed, respectively.

Class 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 some 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 token-matching unit (TMU).

Class predicate_manager. A predicate_manager specifies a predicate which is a set of member functions that are candidates for execution. The

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

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

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

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

Fig. 5. Two uses of result variables. In this example, bar is a regular Mentat class.

compiler constructs a predicate manager 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) [Grimshaw et al. 1990]. MMPS provides C++ class-based naming and message-passing services, message construction, synchronous and asynchronous send, and blocking and nonblocking receive.

An MMPS-name is an address that names a specific object (process) on a specific processor. It contains 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, an MMPS-name contains an index into a host table that contains 30-character host names (IP format) and a port number. On the Intel iPSC/860 and Intel Paragon, an 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. We expect to change this soon, though, to allow objects to migrate to support fault tolerance and better resource utilization.

3.2.2 *Run-Time Dataflow Detection.* The objective of 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 5) 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 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. A strict expression is an expression in which we must have the value in order to proceed. For example, on line 4 of Figure 5(b) the plus operator is strict; we must have the value of w to proceed.

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) ;
```

A *result variable* (RV) is a variable that occurs on the left-hand side of a Mentat assignment statement, e.g., w in Figure 5. It has a delayed value if the most recent assignment statement to it was a computation instance and if 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 data dependence 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 the result variable set. 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 result variable set called the RV_TABLE. There are two cases to consider. If the result variable is not in the RVS then it is not in the RV_TABLE, and its value is already stored at its address in memory. If the result variable address is in the RVS then its value depends on a Mentat expression not yet evaluated, and the result variable is stored in the RV_TABLE. Each RV_TABLE entry contains the result variable's address and a pointer to a computation instance. The computation instance corresponds to the most recent assignment to the variable.

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

```
RV_INSERT((char*)rv_address, computation_instance* node) ;
RV_DELETE((char*) rv_address) ;
force( ) ;
RESOLVE((char*) rv_address, int size) ;
```

The function RV INSERT( ) creates an entry in the RV TABLE for the result variable pointed to by rv_address with a computation instance pointed to by node. 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. RV_INSERT( ) is the mechanism for adding a PRV to the 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( ) examines all of the locally created computation instances. If the computation instance has not previously been evaluated, i.e., its arguments have not been sent, then a future is constructed, and the arguments are sent to the object that is being invoked. Once the arguments have been sent, the computation instance is marked as evaluated. If the computation instance has no actual arguments, then the corresponding member function invocation will receive its arguments from its predecessors in the program graph.

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 until the result is available. Otherwise the value is known and simply returned. Once the result is available, RESOLVE( ) places the result into the memory to which the rv_address points.

In Figure 5 two program fragments were presented to illustrate blocking versus nonblocking member function invocation. The MPL translations for these code fragments are shown in Figures 6(a) and 6(b), respectively. In Figure 6(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 6(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.

This example illustrates the basic concepts used to detect data dependence at run-time. Although 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.

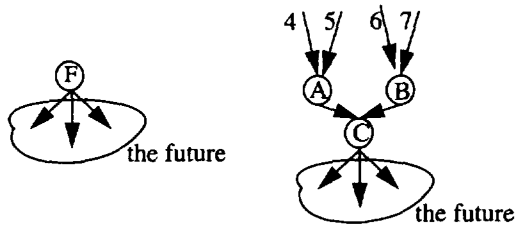
3.2.3 *Token Matching.* In the macro dataflow model, as in pure dataflow, tokens belonging to a particular computation must be matched [Srini 1986; Veen 1986]. For example, in Figure 7(a), the tokens containing the values 4 and 5 must be matched. When both tokens are available, and they have been matched, we say the actor is enabled and may fire (execute). The

```
bar A ;
int w,x,y ;
//  w = A.op1 (4,5) ;
//    Add w to RVS, create a node in the subgraph, marshall arguments.
(*RV_INSERT((&w))) = A.invoke_fn(101,2,     //  function 101 takes two arguments
    ICON_TO_ARG(4),                          //  marshall the first argument
    ICON_TO_ARG(5)) ;                        //  marshall the second argument

//  x = B.op1(6,7) ;
(*RV_INSERT((&x))) = B.invoke_fn(101,2,ICON_TO_ARG(6),ICON_TO_ARG(7)) ;

//  y = C.op1(w,x) ;
(*RV_INSERT((&y))) = C.invoke_fn(101,2,PRV_TO_ARG(&w,sizeof(w)),PRV_TO_ARG(&
x,sizeof(w))) ;

//  rtf(y) ;
//  elaborate the current actor into the constructed subgraph
rtf(PRV_TO_ARG(&y,sizeof(y))) ;
```

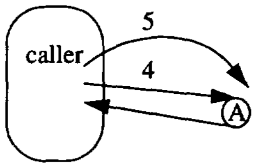
(a) Code transformation for Figure 5(a)

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

//  y = w +1;
y =   (RESOLVE(&w),w) + 1;     //  force( ) and wait for the value
```

(b) Code transformation for Figure 5(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).

Fig. 6. Code transformations and generated graphs for code fragments of Figure 5. ICON_TO_ARG and PRV_TO_ARG are marshalling functions for integer constants and PRVs respectively. PRV_TO_ARG marshalls 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.

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 7(b) there are three + actors and four tokens, two for the first operand of +, 5 and 3, and two for the second operand, 4 and 2. The problem is correctly matching the 5 and 4 to the $+_1$ execution and the 3 and

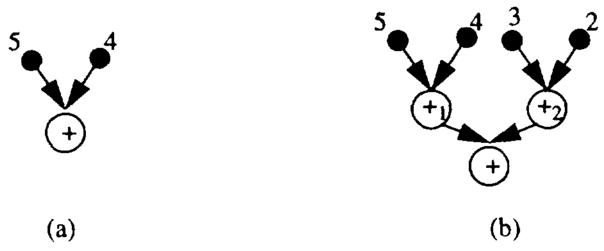


Fig. 7. Token-matching example. An actor may fire when there are tokens on each input arc.

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 dataflow using token coloring [Srini 1986; Veen 1986]. 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 object's local address space. In the latter 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 matched. In this case, the tokens must be matched first by another agent, the token-matching unit. The regular-object token-matching problem is complicated by the fact that the tokens may be 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 *Token Matching for Persistent Objects.* Token matching for persistent objects is performed by the predicate manager (PM). The PM is a library linked into all Mentat applications and objects. In addition to token matching, the PM supports the MPL mselect/maccept statement. For example, the compiler can construct a predicate that says that it is interested in messages for member functions 101, 103, and 105, where 101, 103, and 105 take one, three, and two arguments, respectively. The PM 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

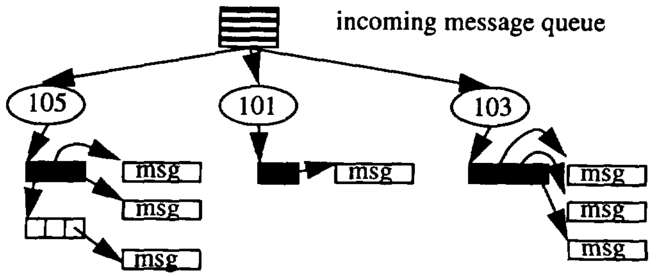


Fig. 8. The predicate manager maintains its message database as a set of linked lists of *work units*. Incoming messages are forwarded to the correct list based on the function number. The complete work units are shaded.

all share the same computation tag, i.e., they correspond to the different arguments of the same instance of a member function invocation. A work unit is complete when all argument messages have arrived.

Work units are stored in a structure known as the *actor list*. The actor list contains an entry for each separate invocation on a member function. This is shown in Figure 8. For example, member function 105 has two entries in its actor list. The work unit corresponding to the first entry is complete. The collection of messages into work units corresponds to token matching in the MDF model.

The predicate management interface to the compiler allows the compiler to specify an ordered list of functions. 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 guards are evaluated to determine the ordered list of functions with the list ordered by priority. This is illustrated in Figure 9.

At run-time when an mselect/maccept is encountered the list of functions is generated, and a call to block_predicate is made. First, 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 is 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 until the list is exhausted. If none is found, block_predicate blocks on message arrival. The code fragment and transformation shown in Figure 9 illustrate how the PM is used to support the mselect/maccept semantics.

3.2.5 *Token Matching for Regular Objects.* 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.

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

(a) MPL mselect/maccept statement. The maccept op2( ) is guarded by (z > 5).

```
{
int pred_number ;
mentat_message *msg[2] ;
predicate_manager pred(2) ;
pred.enable_operation(0, 101, 2) ;
   // The first parameter is the result that will be returned if
   // operation 101 is accepted. The second parameter is the function
   // identifier, and the third indicates the number of arguments to 101.
if (z > 5) pred.enable_operation(1, 102, 2) ;
   // The guard (z > 5) must be true for operation 102 to be enabled.
pred_number = pred.block_predicate(msg) ;
  switch (pred_number)  {
  case 0  :  {  op1(int,int) has been called, service it.
    int arg1 = RESOLVE_MSG(int, msg[0]) ;
    // The macro RESOLVE_MSG converts a message to the indicated type.
    int arg2 = RESOLVE_MSG(int, msg[1]) ;
    op1(arg1,arg2) ;
    delete msg[0] ;
    delete msg[1] ;  }
    break ;
  case 1  :  {  //  op2(int,int) has been called, service it.
    int arg1 = RESOLVE_MSG(int, msg[0]) ;
    int arg2 = RESOLVE_MSG(int, msg[1]) ;
    op2(arg1,arg2) ;
    delete msg[0] ;
    delete msg[1] ;  }
    break ;
  }
}
```

(b) Translation of (a) generated by the MPL compiler. Each enable operation tells the predicate manager that calls on that function are to be accepted.

Fig. 9. Code transformations for run-time support of mselect/maccept.

Token matching is accomplished using a distributed algorithm. The number of TMUs grows with system size so that the token matching does not become a bottleneck. On Unix machines, there is one TMU for each processor in the system. On multicomputers such as the Intel Paragon, a TMU is assigned to each processor cluster. The algorithm is both simple and scalable 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 same TMU. This 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 [Dennis 1975; Srini 1986]. The hash functions are very simple and uniformly distribute the tokens to the TMUs at run-time. 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 PM. 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 an *instantiation* request to the local instantiation manager (IM). The TMU does not block on the instantiation request. Instead, it continues to receive unmatched tokens and issues additional instantiation requests. Thus, several instantiation requests may be outstanding at any given time.

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 member function. The TMU extracts from the token database the tokens for the computation and 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 rematched using the PM, and the member function is executed to completion.

3.2.6 *Scheduling and Instantiation.* The IMs 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., 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 Grimshaw and Vivas [1991].

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 [Casavant and Kuhl 1988; Eager et al. 1986; Hac 1989]. 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) [Grimshaw and Vivas 1991] is heuristic and based upon the work of Eager et al. [1986] 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.

Scheduling decisions are reached using a distributed algorithm. Each node (or virtual node in the case of multicomputers) has an instantiation manager and a token-matching unit. The scheduling decision on each IM consists of two subdecisions: (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 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. Only newly received tasks are eligible for transfer.

In order to avoid instability, where nodes are devoting all of their time to transferring tasks and none of their time to processing them, we employ a simple control policy that places a *transfer limit* on the number of times a task may be transferred. When the transfer limit is reached the IM must accept the task. The transfer limit is set in a configuration file.

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.*

Once the task location is determined, the instantiation of a persistent Mentat object is accomplished by loading a new copy of the executable. For the instantiation of regular objects, the IM tries to *reuse* an existing object of the same class, i.e., it does not load a new instance. Instead it returns the name of the existing object. This saves the time required to perform the load and exploits the fact the regular-object instances are *stateless.*

4. IMPLEMENTATION

As of January, 1995, Mentat has been implemented on 10 platforms. The implementations fall into two categories: Unix-based workstation networks, the Sun 3, Sun 4, Silicon Graphics, Hewlett-Packard, and IBM RS/6000; and distributed-memory MIMD multicomputers, the TMC CM-5, IBM SP-2, Intel iPSC/2,[3] Intel iPSC/860 Gamma, and the Intel Paragon. Within each category the implementations are similar, although there are some significant differences among 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 Paragon.

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 process model, while offering many options, differs in subtle ways from platform to platform.

Mentat objects are implemented by Unix processes that are forked by the IM. Multiple Mentat objects may reside on each Unix host and may execute concurrently. All communication between objects, including intrahost communication, is implemented in MMPS using UDP packets and an interrupt-driven, stop-and-wait protocol.

4.2 Multicomputers

The four multicomputers to which we have ported Mentat differ in several important respects—yet they are very similar in terms of their communication support; all 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 and Paragon, to full process support on the IBM SP-2.

On the CM-5 there is one user process per processor, and all processors must execute the same executable image.[4] Thus, there is no support for dynamically loading different Mentat object executables on different processors, nor for multiple Mentat objects per processor. Instead all Mentat objects, the IM, 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 are the IBM SP-2, Intel iPSC/2, and Paragon, which have multitasking operating systems. The iPSC/2 supports up to 20 processes per processor and the capability to load new executables. Thus, the iPSC/2 execution model is very similar to the Unix model. The Paragon nodes run OSF-1 and have complete multitasking support.

In the middle is the Intel iPSC/860 Gamma. The Gamma supports exactly one process per processor. However, the executable image on the

[3]The iPSC/2 and the CM-5 are currently unsupported.

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

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.

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 low. The performance of each Mentat run-time system component is presented for a network of Sun SparcStation 2 workstations and the Intel Paragon. 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.

We have divided the functions into two types: **primitives** and **composites.** The primitives include communication, dataflow detection, token and predicate management, and scheduling, and correspond to run-time system features already discussed. The composites contain a number of primitive operations and capture all of the overhead terms for three different scenarios. Below are descriptions of the overhead terms, and how they were measured, followed by the data for the two platforms.

4.3.1 *Communication*

—New and delete a mentat_message—Allocate a mentat message, initialize its members, and delete it.

—Message transport—Send a message containing one byte of data. There is a 192-byte header in the Unix implementation and a 124-byte header on the Paragon. 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. The result is then divided by two to determine the one-way cost.

—Message send—Send a one-byte message, but do not 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.

—Communication bandwidth—the per-byte communication bandwidth. Computed as the message transport time for an *N*-byte message minus latency (one-byte message transport) divided by the message size *N*.

All communication times are based on MMPS [Grimshaw et al. 1990]. All times are the average of 1000 iterations.

4.3.2 *Dataflow Detection*

—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 100 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, since the number of entries is typically low.
—RV_TABLE lookup—Look up an entry in the RV_TABLE.

4.3.3 *Token and Predicate Management*

—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 while waiting for a message. However, the time to probe the message system to determine if a message has arrived is included. Two predicates were enabled; the match was found for the first predicate. The cost includes the token matching.

4.3.4 *Scheduling*

—Object instantiation—The time required to create a new Mentat object. This includes all scheduling time and the time to load the executable. Because measurements were done on an idle system these results should be viewed as lower bounds, not expected values. Note that the time will normally vary depending on the number of transfers required for scheduling. Subsequent loads of the same executable from the same instantiation manager are often much faster due to file caching by the operating system.
—Object destruction—The time required to destroy a Mentat object.

4.3.5 *Composite Costs*

—Null RPC on a persistent object—Perform a blocking RPC that takes an integer parameter and returns an integer. This includes the time to construct the graph, transport the tokens, match the tokens at the invoked object, and return the result to the caller. No computation is performed.
—Null RPC on a regular object—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 IM), transport the tokens from the TMU to the object, match the tokens at the object, and return the result to the caller.
—Total Mentat overhead—Null RPC time $-$ $2\times$(message transport time). This captures Mentat overhead versus handwritten send/receive.

```
int i,j ;
timer.start( ) ;
for (i 0;i < 25;i+ +)  {
     j = A.op1(i) ;
     j = B.op1(j) ;
     j = C.op1(j) ;
     j = D.op1(j) ;
     D.sink (j) ;
}
//  wait for last computation
if (D.done( )) ;
timer.stop( ) ;
```

```
int i,j ;
timer.start( ) ;
for (i 0;i < 25;i+ +)  {
     j = A.op1(i) ;
     j = A.op1(j) ;
     j = A.op1(j) ;
     j = A.op1(j) ;
     D.sink(j) ;
}
//  wait for last computation
if (D.done( )) ;
timer.stop( ) ;
```

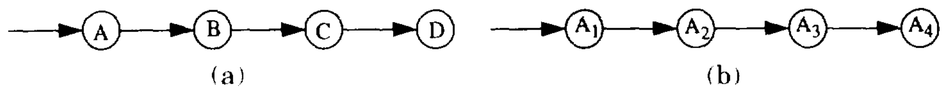


Figure 10

—Pipelined RPC on persistent objects—Four persistent objects are instantiated and then called as shown; see Figure 10(a). The member functions perform no computation and return a value. The average time per member function invocation is determined by dividing the elapsed time by 100, e.g., by 4 $\times$ 25.

—Pipelined RPC on regular objects—This is similar to the above except that A is regular object; see Figure 10(b).

4.3.6 *Environments*

Sun SparcStation 2. The Sun configuration consisted of a collection of 8 Sparc2s connected by ethernet. Each Sparc (Sun 4) processor runs at 40MHz and was configured with 32MB of real memory. The run-time system performance is presented in Table I.

Intel Paragon. The Paragon performance data was collected on a 56-node Intel Paragon at the Jet Propulsion Laboratory using an 8-node partition. Each i860 processor runs at 40MHz and contains 32MB of real memory. The run-time system performance is also shown in Table I.

4.4 Performance Observations

On the Sun network, the 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 RVs, creating computation instances, adding arcs, and constructing short future lists is quite small (120μsec.) when compared to the message transport costs. This number is obtained by adding up the component costs for these operations in Table I.

On the Intel Paragon, 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 Paragon has an optimized communication system and special communication hardware. The result is

Table I. Performance of Mentat RTS on the Sun Network and Intel Paragon

		Cost (μsec.)	
Category	Functions	Sun Network	Intel Paragon
Communication	New and delete a message	16	16
	Message transport	1730	260
	Message send	566	16*
	Bandwidth (KB/sec.)	710	3930
Dataflow Detection	New and delete computation instance	21	15
	Add arc	3	3
	Construct a future—short	57	6
	Construct a future—100 actors	1684	910
	RV_Table insert	13	20
	RV_Table lookup	10	20
Token Management	Block predicate	170	48
Scheduling	Object instantiation	235,800	621,000
	Object destruction	2400	300
Composites	Single null RPC—persistent	3960	1280
	Pipelined null RPC—persistent	775	413
	Single null RPC—regular	12,320	520,800
	Pipelined null RPC—regular	2950	10,210
	Total Mentat overhead	500	760

*The latency is for a zero-byte Mentat message that includes over 100 bytes of header and is therefore higher than the minimum latency reported elsewhere.

that the RTS software overhead, which remains fairly constant between these two platforms, is the dominant overhead on the Paragon. On both platforms, object instantiation is also very slow due to the overhead of process creation in these environments—fork/exec in Unix and load in NX. The cost for regular-object RPC and pipelining includes the cost of object instantiation while the cost for persistent object RPC and pipelining does not. Hence, the cost of these operations for regular objects is higher. On the Paragon the cost of regular-object RPC is even higher, since there is currently no regular-object reuse on this platform.

With low-level performance numbers in hand we can now address the question: is Mentat efficient? Mentat introduces overhead that will not be experienced by a hand-coded implementation. This raises several new questions. What is the penalty versus hand-coded? Given the overhead costs what are the granularity requirements for Mentat objects? And finally, are there any performance benefits obtained by using Mentat? Below we answer these in turn.

What is the Performance Penalty? From Table I we see that the cost of performing an RPC "by hand" on the Suns is 2 × (message transport + new and delete message) = 2 × (1730 + 31) = 3522μsec. The Mentat time is 3960μsec., a 12% increase. Whether this is acceptable depends on the application, though even a hand-coded application must be fairly coarse grain to achieve good performance. On the Paragon the picture is worse. The "by-hand" time is 2 × (260 + 16) = 552μsec. versus 1280μsec. for Mentat, a 132% increase in overhead. This means that the minimum grain size for Mentat applications is significantly larger on the Paragon than for hand-coded applications. This is significant for applications that require a

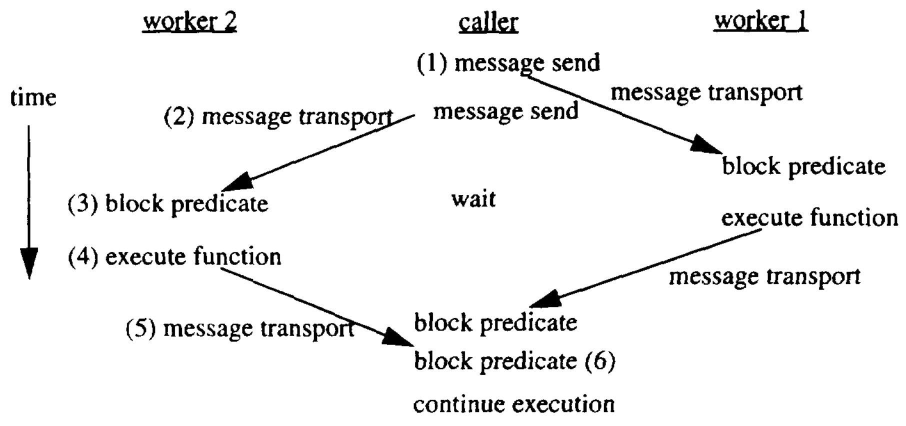


Fig. 11. Timeline for two-worker fan-out/fan-in.

small grain size. As to inherently fine-grained problems, it is important to note that the ratio between processor performance (MFLOPS) and interprocessor communication bandwidth is expected to increase over time, making inherently fine-grained problems more difficult to execute on MIMD machines.

What are the Granularity Requirements for Mentat Applications? The granularity requirements depend on the Mentat application. To give some insight into this problem, we consider a classic fan-out/fan-in computation with k workers implemented as a k-ary tree. This is a simple scenario that we use to determine the magnitude of the minimum granularity. More complex computations with a less regular structure would require a more detailed critical-path analysis, but would likely be within an order of magnitude of this value. To determine the granularity requirements for this problem, we compute the minimum grain size needed to break even. The break-even point is where the sequential time and parallel time are equal, that is, where the benefits of parallel execution begin to offset the overhead. Beyond this point, it is profitable to exploit parallelism. The minimum acceptable granularity should not be confused with the desired computation granularity. Mentat applications often exhibit a computation granularity much higher than the minimum in order to obtain the best possible performance. We will assume that we have idle processors and an idle network, i.e., that other users will not impact performance.

In Figure 11, we show a timeline for the two-worker case with the major cost components included. The minimum number of workers needed to realize any performance gain if all real work is done by the workers is two; if one worker is used then we pay all of the overhead and receive none of the benefit of parallel execution. The critical path contains the labeled primitives. (For the following equations, we use the following abbrevia-

Table II. Minimum Granularity on the Sun Network and Intel Paragon

	Function Time in μsec.		Grain Size in *fp* Instructions	
Environment	k=2	k=4	k=2	k=4
Sun Sparcstation 2	4356	1823	30492	12761
Intel Paragon	676	265	6760	2650

tions: MS = message send time; MT = message transport time; FT = function execution time; and BP = block predicate time.) The total execution time is the time on the critical path:

$$\underset{(1)}{\mathrm{MS}} + 2 \times \underset{(2,\ 5)}{\mathrm{MT}} + \underset{(4)}{\mathrm{FT}} + 2 \times \underset{(3,\ 6)}{\mathrm{BP}}$$

The asynchronous message sends are sequential from the caller with the last message send included in message transport cost (2). If we assume that the function time is the same for both workers, then break-even is achieved for two workers when:

$$2 \times (\mathrm{FT}) = \mathrm{MS} + 2 \times (\mathrm{MT} + \mathrm{BP}) + \mathrm{FT}$$

The left-hand side is the sequential time, and the right-hand side is the parallel time computed above. Solving for FT yields:

$$\mathrm{FT} = \mathrm{MS} + 2 \times (\mathrm{MT} + \mathrm{BP})$$

This generalizes to k workers; break-even is achieved when:

$$k \times (\mathrm{FT}) = k - 1 \times \mathrm{MS} + 2 \times (\mathrm{MT} + \mathrm{BP}) + \mathrm{FT}$$

$$\mathrm{FT} = (k - 1 \times \mathrm{MS} + 2 \times (\mathrm{MT} + \mathrm{BP}))/(k - 1)$$

This analysis makes the simplifying assumption that message transport cost is independent of the number of workers. Often this is not the case due to contention for communication bandwidth. The product of the function time and the processor rate (in MFLOPS) gives the minimum granularity of the break-even point in units of floating-point instructions. The Sun 4 used in the tests is a 7-MFLOP machine, and the i860 is an 80-MFLOP peak performance processor. The i860 peak is not realizable on typical codes. A rate of 10 MFLOPS on the i860 is considered good [Moyer 1991]. We assume 7 and 10 MFLOPS for the Sun 4 and i860, respectively. Using the data from Table I and these floating-point rates we obtain the function time and minimum grain sizes for $k = 2$ and $k = 4$ as shown in Table II.

The table values are computed easily. Consider $k = 2$ for the Sun network. The minimum function granularity for the Sun 4 is:

$$FT = 556 + 2 \times (1730 + 170) = 4356\mu\text{sec}.$$

Since the Sun 4 is a 7-MFLOP machine, the minimum grain size in floating-point operations is 7 × 4356 or 30,492 floating-point operations. The other table values are computed in a similar fashion. As expected, the minimum granularity is much larger for the Sun 4 due to the high communication cost on the network.

Performance Benefits. A final comment on performance is in order. Mentat performance is not always worse than hand-coded. On the contrary, we have observed that Mentat performance can be superior to hand-coded performance. At first this might seem counterintuitive. The reason is that Mentat often exploits concurrency that a programmer may not attempt to exploit because of the increased complexity.

Two examples illustrate this. First, there are data-parallel applications that iteratively perform a work phase, in which an operation is carried out in parallel on the data set, and a reduction phase, where the results are, for example, sorted and merged. The natural way to code this by hand is to perform the data-parallel operation for the *i*th iteration, collect the results, perform the next data-parallel operation, and so on. Note the synchronization point at the end of each iteration. Often the Mentat code will not synchronize at the end of each iteration. Instead, the compiler will generate code that results in the pipelined execution of the data-parallel operation and the reduction phase. Thus, the data-parallel operation and the reduction are performed concurrently. Further, if the data-parallel operations require a data-dependent, variable amount of time then the workers will finish at different times. In a hand-coded implementation the next iteration begins only when all workers have completed. Once again, the Mentat code may be less synchronous, synchronizing only at the end of all iterations. Thus if different workers are slower for different iterations, then the less synchronous code will complete more quickly.

A second example is pipelined data filtering and feature extraction applications. When constructing pipelined applications by hand a great deal of effort goes into balancing the amount of work performed in each stage because the throughput is limited by the slowest stage. If the computation requirements for a stage are data dependent then balancing the work by hand is difficult, and often not performed. This can be overcome by using regular Mentat objects. The system instantiates a new instance for each instance of each stage. Thus, if one iteration is slower, additional processor resources are utilized so that the next iteration may begin as soon as the data is available, keeping the pipe full and preventing a “bubble” from forming. While this can be done by hand, it often is not.

5. RELATED WORK

Work related to our effort falls into four categories: other object-oriented parallel processing systems, other compiler-based distributed-memory systems, other portable parallel processing systems, and other large-grain dataflow systems.

5.1 Parallel Object-Oriented Systems

In the object-oriented parallel processing domain Mentat differs from systems such as Parmacs [Beck 1990], Presto [Bershad et al. 1988], and Jade [Lam and Rinard 1991] (shared-memory object-based systems) in its ability to easily execute on both shared-memory MIMD and distributed-memory MIMD architectures, as well as hybrids. pC++ [Bodin et al. 1993] and Paragon [Cheung and Reeves 1992] on the other hand are data-parallel derivatives of C++. Mentat accommodates both functional and data parallelism, often within the same program. ESP [Smith et al. 1989] is perhaps the most similar of the parallel object-oriented systems. It, too, is a high-performance extension to C++ that supports both functional and data parallelism. What distinguishes Mentat is our compiler support. In ESP, remote invocations either return values or futures. If a value is returned, then a blocking RPC is performed. If a future is returned, it must be treated differently. Futures may not be passed to other remote invocations, limiting the amount of parallelism. Finally, ESP supports only fixed-size arguments (except strings). This makes the construction of general-purpose library classes, e.g., matrix operators, difficult.

There are also a large number of object-based systems and languages for distributed systems [Bal et al. 1989; Chin and Chanson 1991], as opposed to parallel systems. Mentat differs from these and other distributed object-based systems in our objectives: we strive for performance via parallelism rather than distributed execution.

5.2 Compiled Distributed-Memory Systems

Until recently, there were few results for compiled—as opposed to hand-coded—applications on distributed-memory machines. There are now several active projects in this area, Fortran-D [Fox et al. 1990], HP Fortran [Loveman 1993], Dataparallel C [Nedeljkovic and Quinn 1992], Paragon [Cheung and Reeves 1992], and the Inspector/Executor [Wu et al. 1991] model, to name a few. These are primarily data-parallel languages with Wu et al. tied to a data-parallel model of computation. What differentiates our work from theirs is that Mentat exploits opportunities for both functional and data parallelism. Further, in Mentat, parallelism, and the communication and synchronization constructs that are required, is dynamically detected and managed. Most other systems statically determine the communication and synchronization requirements of the program at compile-time.

5.3 Portable Systems

Applications portability across parallel architectures is an objective of many projects. Examples include PVM [Sunderam 1990], Linda [Carriero and Gelernter 1989; Carriero et al. 1992], the Argonne P4 macros [Boyle et al. 1987], Fortran D [Fox et al. 1990], and Parti [Mirchandaney et al. 1988].

Our effort shares with these and other projects the basic idea of providing a portable virtual machine to the programmer. The primary difference is the level of the abstraction. Low-level abstractions (e.g., send/receive and in/out) such as in Boyle et al. [1987], Carriero and Gelernter [1989], and Sunderam [1990] require the programmer to operate at the assembly language level of parallelism. This makes writing parallel programs more difficult. Others [Beguelin et al. 1992; Fox et al. 1990; Hatcher et al. 1991; Shu and Kale 1991] share our philosophy of providing a higher-level language interface in order to simplify applications development. Parti is limited to run-time data-level parallelism. What differentiates our work from other high-level portable systems is that we support both functional and data parallelism as well as support the object-oriented paradigm.

5.4 Large-Grain Dataflow

Mentat and macro dataflow (MDF) differ from other coarse-grain dataflow systems such as CDF [Babb 1984], HENCE [Beguelin et al. 1992], and Code/Rope [Browne et al. 1990] in several ways. First, MDF graphs are dynamic rather than static. Second, some MDF actors maintain state internally, rather than copying and circulating tokens to maintain state. Third, the MDF program graphs are constructed at run-time using compiler-generated code. Thus, the programmer is not responsible for generating the program graphs using a graphical interface as in Beguelin et al. and Browne et al. Finally, the MDF model has been implemented on *both* shared-memory and distributed-memory MIMD machines.

6. SUMMARY AND FUTURE WORK

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, dynamic data dependence detection and management, 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. To date, the run-time system has been implemented on platforms that include networks of Sun 3s, Sun 4s, HPs, IBM RS/6000s, and Silicon Graphics workstations, as well as multicomputers such as the Intel iPSC/860, Intel Paragon, and the IBM SP-2.

We presented the overhead costs for the dominant run-time services on two platforms: the Sun SparcStation 2 and an Intel Paragon. Performance on applications is good [Grimshaw 1993b; Grimshaw et al. 1993a; 1993b] for problems of sufficient granularity and leads us to the conclusion that the run-time system overhead is tolerable for many applications. However, the overhead is too high for those applications that require fine-grain communication or that have been coded to require fine-grain communication.

Future work on the Mentat run-time system falls into three categories: porting to additional platforms, generating a thread-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 the DEC alpha. In the multicomputer domain we are targeting the Cray T3D. 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: intrahost 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. This will enable Mentat to support more efficient execution of finer-grained applications. A significant disadvantage is that many user-defined library routines are not reentrant. 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 [Grimshaw et al. 1994] 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 based on computation granularity and application communication topology [Weissman and Grimshaw 1994].

ACKNOWLEDGMENTS

This research was performed in part using the Intel Gamma and Paragon operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the Jet Propulsion Laboratory.

REFERENCES

ABELSON, H., SUSSMAN, G. J., AND SUSSMAN, J. 1985. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge Mass.

AGERWALA, T. AND ARVIND. 1982. Data flow systems. *IEEE Comput. 15,* 2 (Feb.), 10–13.

BABB, R. F. 1984. Parallel processing with large-grain data flow techniques. *IEEE Comput. 17,* 7 (July), 55–61.

BAL, H., STEINER, J., AND TANENBAUM, A. 1989. Programming languages for distributed computing systems. *ACM Comput. Surv. 21,* 3 (Sept.), 261–322.

BECK, B. 1990. Shared memory parallel programming in C++. *IEEE Softw.* 7, 4 (July), 38–48.

BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. 1988. Presto: A system for object-oriented parallel programming. *Softw. Pract. Exper. 18*, 8 (Aug.), 713–732.

BEGUELIN, A., DONGARRA, J., GEIST, A., MANCHECK, R., AND SUNDERAM, V. S. 1992. HeNCE: Graphical development tools for network-based concurrent computing. In *Proceedings of SHPCC-92*. IEEE Computer Society Press, Los Alamitos, Calif., 129–136.

BODIN, F., BECKMAN, P., GANNON, D., NARAYANA, S., AND YANG, S. 1993. Distributed pC++: Basic ideas for an object parallel language. In *Proceedings of the Object-Oriented Numerics Conference*. Rogue Wave Software, Corvallis, Oreg.

BROWNE, J. C., LEE, T., AND WERTH, J. 1990. Experimental evaluation of a reusability-oriented parallel programming environment. *IEEE Trans. Softw. Eng. 16*, 2 (Feb.), 111–120.

CARRIERO, N. AND GELERNTER, D. 1989. Linda in context. *Commun. ACM 32*, 4 (Apr.), 444–458.

CARRIERO, N., GELERNTER, D., AND MATTSON, T. G. 1992. Linda in heterogeneous computing environments. In *Proceedings of the WHP92 Workshop on Heterogeneous Processing*. IEEE, New York, 43–46.

CASAVANT, T. L. AND KUHL, J. G. 1988. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng. 14*, 2 (Feb.), 141–154.

CHIN, R. AND CHANSON, S. 1991. Distributed object-based programming systems. *ACM Comput. Surv. 23*, 1 (Mar.), 91–127.

CHEUNG, A. L. AND REEVES, A. P. 1992. High performance computing on a cluster of workstations. In *Proceedings of the 1st Symposium on High-Performance Distributed Computing*. IEEE Computer Society Press, Los Alamitos, Calif., 152–160.

DENNIS, J. 1975. First version of a data flow procedure language. MIT TR-673, MIT, Cambridge, Mass. May.

EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. 1986. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng. 12*, 5 (May), 662–675.

FOX, G. C., HIRANANDANI, S., KENNEDY, K., KOELBEL, C., KREMER, U., TSENG, C. W., AND WU, M. Y. 1990. Fortran D language specifications. Tech. Rep. SCCS 42c, NPAC, Syracuse Univ., Syracuse, N.Y.

GRIMSHAW, A. S. 1993a. The Mentat computation model—data-driven support for dynamic object-oriented parallel processing. Computer Science Tech. Rep. CS-93-30, Univ. of Virginia, Charlottesville, Va. May.

GRIMSHAW, A. S. 1993b. Easy to use object-oriented parallel programming with Mentat. *IEEE Comput. 26*, 5 (May), 39–51.

GRIMSHAW, A. S. AND VIVAS, V. E. 1991. FALCON: A distributed scheduler for MIMD architectures. *In Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*. USENIX Assoc., Berkeley, Calif., 149–163.

GRIMSHAW, A. S., MACK, D., AND STRAYER, W. T. 1990. MMPS: Portable message passing support for parallel computing. In *Proceedings of the 5th Distributed Memory Computing Conference*. IEEE Computer Society Press, Los Alamitos, Calif., 784–789.

GRIMSHAW, A. S., STRAYER, W. T., AND NARAYAN, P. 1993b. Dynamic object-oriented parallel processing. *IEEE Parallel Distrib. Tech. Syst. Appl. 1*, 2 (May), 33–47.

GRIMSHAW, A. S., WEISSMAN, J. B., WEST, E. A., AND LOYOT, E. 1994. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *J. Parallel Distrib. Comput. 21*, 3 (June).

GRIMSHAW, A. S., WEST, E. A., AND PEARSON W. R. 1993a. No pain and gain!—Experiences with Mentat on biological application. *Concurrency Pract. Exper. 5*, 4 (July), 309–328.

HAC, A. 1989. Load balancing in distributed systems: A summary. *Perf. Eval. Rev. 16*, (Feb.), 17–25.

HALSTEAD, R. H., JR. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct.), 501–538.

HATCHER, P. J., QUINN, M. J., LAPADULA, A. J., SEEVERS, B. K., ANDERSON, R. J., AND JONES, R. R. 1991. Data-parallel programming on MIMD computers. *IEEE Trans. Parallel Distrib. Syst. 2*, 3, 377–383.

INTEL. 1988. *iPSC/2 USER'S GUIDE*. Intel Scientific Computers, Beaverton, Oreg.

LAM, M. S. AND RINARD, C. 1991. Coarse-grain parallel programming in Jade. In the *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, 94–105.

LARUS, J. R., RICHARDS, B., AND VISWANATHAN, G. 1992. C**: A large-grain, object-oriented, data-parallel programming language. UW Tech. Rep. 1126, Computer Sciences Dept., Univ. of Wisconsin, Madison, Wisc. Nov.

LEE, J. K. AND GANNON, D. 1991. Object oriented parallel programming experiments and results. In *Proceedings of Supercomputing '91*. IEEE Computer Society Press, Los Alamitos, Calif., 273–282.

LOVEMAN, D. B. 1993. High performance Fortran. *IEEE Parallel Distrib. Tech. Syst. Appl. 1*, 1 (Feb.), 25–42.

LUSK, E. L. AND OVERBEEK, R. 1987. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, New York.

MACCALLUM, L. AND GRIMSHAW, A. S. 1994. A parallel object-oriented linear algebra library. In *Proceedings of the Object-Oriented Numerics Conference*. Rogue Wave Software, Corvallis, Oreg., 233–249.

MIRCHANDANEY, R., SALTZ, J. H., SMITH, R. M., NICOL, D. M., AND CROWLEY, K. 1988. Principles of runtime support for parallel processors. In the *International Conference on Supercomputing*. ACM, New York.

MOYER, S. A. 1991. Performance of the IPSC/860 Node Architecture. Computer Science Tech. Rep. IPC91-09. IPC, Charlottesville, Va. May.

NEDELJKOVIC, N. AND QUINN, M. J. 1992. Data-parallel programming on a network of heterogeneous workstations. In *Proceedings of the 1st Symposium on High-Performance Distributed Computing*. IEEE Computer Society Press, Los Alamitos Calif., 28–36.

SHU, W. AND KALE, L. V. 1991. Chare kernel—A runtime support system for parallel computations. *J. Parallel Distrib. Comput. 11*, 198–211.

SMITH, S. K., SMITH, R. J., II, CALDWELL, G. S., PORTER, C., LEDDY, W. J., KHANA, A., CHATTERJEE, A., HUNG, Y. T., HAHN, D. W., AND ALLEN, W. P. 1989. Experimental systems project at MCC. Tech. Rep. ACA-ESP089-89, MCC, Austin, Tex.

SRINI, V. P. 1986. An architectural comparison of dataflow systems. *IEEE Comput. 19*, 3 (Mar.), 68–88.

STROUSTRUP, B. 1988. What is object-oriented programming? *IEEE Softw. 5*, 3 (May), 10–20.

SUNDERAM, V. S. 1990. PVM: A framework for parallel distributed computing. *Concurrency Pract. Exper. 2*, 4 (Dec.), 315–339.

WEISSMAN, J. B. AND GRIMSHAW, A. S. 1994. Network partitioning of data parallel computations. In *Proceedings of the 3rd International Symposium on High-Performance Distributed Computing*. IEEE Computer Society Press, Los Alamitos, Calif., 149–156.

WEISSMAN, J. B., GRIMSHAW, A. S., AND FERRARO, R. 1994. Parallel object-oriented computation applied to a finite element problem. *Sci. Comput. 2*, 133–144.

WEST, E. A. AND GRIMSHAW, A. S. 1995. Braid: Integrating task and data parallelism. In *Proceedings of Frontiers '95*.

VEEN, A. H. 1986. Dataflow machine architecture. *ACM Comput. Surv. 18*, 4 (Dec.), 365–396.

WU, J., SALTZ, J., BERRYMAN, H., AND HIRANANDANI, S. 1991. Distributed memory compiler design for sparse problems. ICASE Rep. 91-13, ICASE, Hampton, Va. Jan.

Received July 1993; revised April 1994; accepted October 1994