The Mentat Computation Model Data-Driven Support for Object-Oriented Parallel Processing

Andrew S. Grimshaw

Technical Report No. CS-93-30
May 28, 1993

This work was partially supported by NASA grant NAG-1-1-1181 and NSF grant ASC-9201822.

The Mentat Computation Model - Data-Driven Support for Dynamic Object-Oriented Parallel Processing[1]

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

Abstract

Mentat is an object-oriented parallel processing system developed at the University of Virginia which has been ported to a variety of MIMD architectures. The computation model employed by Mentat is macro data-flow (MDF), a medium grain, scalable, data-driven computation model that supports both high degrees of parallelism and the object-oriented paradigm. A key aspect of the model is that it can be efficiently implemented. Inspired by data-flow, MDF retains the graph-based, data-driven, self-synchronizing aspects of data-flow. MDF address the shortcomings that data-flow exhibits when applied to distributed memory MIMD architectures by extending data-flow in three ways: (1) it is medium grain - actors are of sufficient computational complexity to amortize overhead costs, (2) program graphs are dynamically constructed at run-time - this permits dynamic function binding as required by the object-oriented paradigm and increases the average computation granularity, and (3) actors may maintain state between executions - this provides an effective way to model the state properties of imperative, object-oriented programs. In this paper we present the macro data-flow model, key features of the Mentat Programming Language (MPL), key features of the Mentat run-time system that implements a virtual macro data-flow machine, and translations from MPL constructs to the virtual machine. The result of the translation process is programs in which program graphs are constructed at run-time by observing data dependencies that develop as execution unfolds. These program graphs are then executed in parallel by the run-time system, resulting in improved performance over sequential execution.

1. This work was partially supported by NASA grant NAG-1-1181 and by NSF grant ASC-9201822.

1. Introduction

Writing application software for parallel and distributed systems has proven to be far more difficult than building the hardware. Research in the computer science community has focused on the right languages and elegant abstractions for writing concurrent programs, but the fact remains that today most applications are written by hand using C or Fortran extended with library calls that clearly reflect the underlying hardware. Applications scientists, in our experience, have not embraced the elegant abstractions. These programmers are familiar with the imperative style of programming, Fortran and C in particular, and are not eager to change unless there is a significant benefit. Since high performance is the goal they will not sacrifice, they avoid abstractions, however elegant, for fear they will fail to satisfy their performance needs.

Writing parallel programs by hand is difficult. The programmer must manage communication, synchronization, and scheduling of tens to thousands of independent processes. The burden of correctly managing the environment often overwhelms programmers, and requires a considerable investment of time and intellectual energy. Also, once implemented on a particular MIMD architecture, the resulting codes are usually not usable on other MIMD architectures; the tools, techniques, and library facilities used to parallelize the application are specific to a particular platform. Thus, considerable effort must be re-invested to port the application to a new architecture. Given the plethora of new architectures and the rapid obsolescence of existing architectures, this represents a continuing time investment, and discourages users from parallelizing their code in the first place.

Solutions to the problem of generating parallel software have ranged from fully automatic, compiler-based approaches[2][24][28], to hand-coded, explicit approaches [5][10][11][31][23][26]. The problem with fully automatic compiler-based approaches is that they are good at finding fine-grain parallelism but do not work well for distributed memory machines. Further, they can often be defeated by spurious dependencies that require programmer intervention to resolve, making them less automatic and increasing the cognitive burden on the programmer. The problem with explicit approaches is that they place too large a burden on the programmer. The programmer is responsible for problem decomposition (code and data), scheduling, communication, and synchronization. This task is too complex for all but the most regular of applications. Explicit approaches also suffer from timing dependent errors and Heisenbugs.[1]

If the parallel software problem is not solved, parallel computers will not come into widespread use, rather they will be used only by those organizations that must have the highest performance and are willing to pay for it with programmer time. This would be an unfortunate outcome as there are many other users that could gain from the use of parallel processing, particularly on networks of workstations.

Our solution to the parallel software problem lies in a compromise between compiler-based approaches and manual approaches. Mentat, an object-oriented, medium grain, parallel processing system developed at the University of Virginia, strives for such a compromise. Mentat addresses the parallel software problem by leveraging the object-oriented paradigm and exploiting the comparative advantages of both humans and compilers: people know the application domain and can better decompose the problem, compilers can better manage data dependence and synchronization. Using the Mentat programming language (MPL), an extension to C++, the programmer uses application knowledge to specify the classes whose member functions are sufficiently computationally complex to warrant parallel execution. These computationally "heavy" classes are known as *Mentat classes*. We have extended the object-oriented concept of encapsulation to include parallelism encapsulation both within Mentat class instance member function invocations and between instance member function invocations. Once the Mentat classes have been specified by the programmer, the compiler and run-time system take over and manage scheduling, communication, and synchronization between Mentat objects, and dynamically construct program graphs by observing the data dependencies that actually occur as program execution unfolds.

Mentat also addresses the problem of applications portability by providing source code portability between supported platforms as long as no architecture or operating system-specific features are used.[2] Mentat achieves portability by using a portable virtual machine as a target for the compilation process. The virtual machine is itself relatively easy to port. All architecture-specific features

1. A Heisenbug is a timing dependent bug that goes away when debugging or tracing is turned on. They are frustrating and difficult to track down.
2. Application code may benefit from changes, but does not *require* them, e.g., on the Sun 3/60 loop unrolling provides no benefit; on the SparcStation it does.

have been isolated into two modules. Mentat has been ported to six platforms, including networks of Sun 3's, Sun 4's, Silicon Graphics Irises, the Intel iPSC/2, the Intel iPSC/860, and the TMC CM-5.

The high-level and applications performance aspects of Mentat have been presented elsewhere [14][15][16]. In [14] we present an overview, the Mentat philosophy, the Mentat approach to parallel computing, and performance results. In [16] the performance of a range of applications with a range of speedups are explored; the results are promising, speedups are not only good -- they are competitive with hand-coded versions of the same applications.

The purpose of this paper is to present two of the conceptual underpinnings upon which Mentat rests, the Mentat computation model, macro data-flow (MDF), and the mapping from the MPL to macro data-flow. MDF is a medium grain, data-driven computation model that supports the object-oriented paradigm. It is scalable and can be efficiently implemented on a variety of parallel architectures that includes distributed memory MIMD machines as diverse as networks of workstations and mesh and hypercube-based multicomputers. The mapping from MPL to MDF consists of a set of translations from the key constructs of MPL to run-time system calls that implement the MDF. These calls result in the run-time construction and elaboration of program graphs.

We begin our discussion with some background on computation granularity and compiler techniques for parallelism detection. The MDF model is then presented in detail, followed by a brief overview of the salient aspects of the language, their translation, and the program graph construction techniques used. We do not include a description of run-time issues in this paper. Interested readers should see [17][18] for a description of the run-time system.

2. Background

In order to achieve high performance on a distributed memory MIMD architecture four key issues must be addressed.

1) What size is appropriate for the pieces into which to decompose the problem, i.e., what is the appropriate granularity? This varies from architecture to architecture.

2) How is the application decomposed? This includes not just the computational aspects, but the data decomposition and resulting locality as well.

3) How are the computational and data components scheduled or placed on the processors? This often involves a complex trade-off between load balance and communication overhead. In general, finding an optimal solution is NP-hard.

4) How is the required communication and synchronization between the application components established and managed? This can be difficult and subtle if done by hand.

For an in-depth discussion of these issues, see [3][11][25][29].

It is often claimed that the difficulty in parallel computing is in finding the parallelism in the application. A tool is then presented that in simulation exposes and exploits fantastic levels of parallelism, where hundreds or thousands of concurrent activities are plotted. In our experience though, finding parallelism is not the problem. Most of the applications we have examined have many opportunities for parallel execution. Rather the problem is constraining parallelism and finding the right amount of parallelism for the target architecture given its associated overhead.

Overhead is the friction of parallel processing. It can take many forms not present in sequential programs. Data communication and synchronization between computational units, scheduling, migration, and context switching are among the most costly. All consume the most precious commodity, time. The degree of parallelism and overhead come together in the *granularity of computation*. The granularity of computation is often defined as the number of instructions between overhead events. This leads to the concept of the *computation ratio*. The computation ratio is the ratio of computation time to total time spent in overhead. If the computation ratio becomes small, i.e., less than one, then more time is being spent on overhead than on useful work. In fact it may take longer to solve the problem in parallel than sequentially if the computation ratio is too low. Increasing parallelism can mean poorer performance.

Thus the appropriate granularity depends on overhead. Software overhead can run from a few instructions to tens of thousands. On many multitasking systems, context switch times are on the order of one millisecond. Communication software overhead using IP datagrams is on the order of a few milliseconds. In one millisecond, tens of thousands of instructions can be executed. Thus, millisecond range software overhead argues for the use of medium granularity; using small granularity computation is expensive.

The problem, then, in writing parallel software is decomposing the application into appropriate size granules of computation, scheduling those granules, and managing the communication and synchronization between those granules. Historically, this has been done either using fully automatic parallelizing compilers [2][24][28], or by hand using primitives such as load, send, and receive[8][13][23][31].

There are several advantages of automatic techniques. Ideally, by using these techniques, dusty-deck programs can be parallelized with a minimum of effort. We say “ideally” because it does not always work, and often results in low levels of parallelism due to dependencies and/or a small granularity of computation. Second, problem decomposition, scheduling, communication, and synchronization are all handled by the compiler, an entity we trust, rather than by more error-prone human programmers. Third, semantic equivalence to the original program can be maintained. There are, however, disadvantages with automatic techniques. They are static, looking at the textual representation of the program without any knowledge of the dynamic behavior of the program, e.g., which branches are taken, how many times a loop must be unrolled, what the array index values are (particularly via indirect arrays), etc. This lack of knowledge is a serious impediment. Frequent run-time synchronization is often required to determine which path through the program is actually chosen, thus reducing granularity. Further, automatic techniques must be conservative and often cannot exploit all available parallelism because of spurious dependencies that require programmer intervention to resolve. They cannot take a global view of programs and partition them appropriately because they lack domain knowledge of the application. Automatic techniques are best at finding fine-grain and loop-level parallelism. For this reason they have primarily been used for architectures with very low hardware overhead, e.g., shared memory multiprocessors.

Fully explicit (manual) approaches are often characterized by some sequential language, such as C or Fortran, extended with library calls to support problem partitioning (e.g., light weight threads), communication (e.g., shared memory or messages), and synchronization (e.g., semaphores and barriers).

The advantages of explicit approaches are that they are easy to engineer, they can reflect the underlying hardware, and the programmer has total control over the computation and can partition and schedule the problem based on his domain knowledge. The disadvantages are that the programmer is responsible for all aspects of parallelization, scheduling, communication, and synchronization, and can easily make errors that are timing dependent and difficult to track down. Application portability also suffers because fairly low-level, system-specific features are used.

In summary, the use of medium-grain parallelism permits us to maintain sufficient computation granularity to exploit a wide range of architectural platforms, especially platforms with lower cost (slower) interconnection networks. To easily write medium-grain software calls for the safety, correctness, and ease-of-use of compiler based approaches, combined with the granularity control of explicit approaches. Such a system would permit the exploitation of MIMD architectures by a wide range of users and require a reduced level of programmer effort, effectively moving parallel processing out of the lab and into the marketplace.

3. The Macro Data-Flow Model

Macro data-flow was inspired by the data-flow model of computation [1][12][32]. Recall that in the data-flow model there are two types of objects: *tokens* and *actors*. Tokens carry data or control information. Each actor performs a function based on the information contained in the tokens it consumes. Actors are computation primitives, corresponding to granules of computation. Actors do not preserve their internal state from one computation to the next. In traditional data-flow actors are small grain, for example, add, subtract, multiply, and compare.

A computation can be described by a data-flow graph in which nodes are actors. Actors are connected by directed arcs along which tokens flow. Arcs model data dependencies between granules of computation. Specifically, the execution of any granule cannot begin until all granules of computation on which it depends have completed. Correspondingly, in the model, an actor is enabled and may “fire” (execute) only when there are tokens on all of its incoming arcs. Thus, synchronization is automatically provided by the token transport mechanism. Parallelism is gained in data-flow architectures by allowing any actor to execute on any processor and by allowing as many enabled actors to fire as there are processors to execute them. When there is a sufficiently large number of processors, only actors that depend on uncompleted granules of computation are not enabled. A key feature of the model is that the order of actor execution does not effect the result. Thus, the data-flow model naturally achieves high degrees of parallelism.

3.1 Problems with Data-flow

There are two problems with traditional data-flow with respect to our requirements. First, the granularity of traditional data-flow is too small for many MIMD architectures, particularly distributed memory systems where latencies are measured in hundreds to thousands of microseconds. The overhead of token transport and actor scheduling and instantiation requires that the granularity of computation be at least hundreds, and perhaps thousands, of instructions. Second, traditional data-flow systems have program graphs with a topology that is fixed at compile-time. Even in so-called dynamic graph systems, “dynamic” refers to the dynamic creation of a subgraph instance (for example, a new loop iteration or procedure call). The subgraph structure is still fixed.

We require dynamic topology program graphs for two reasons. Static topologies are inadequate in an object-oriented system because a static topology implies that the types of objects can be determined at compile-time, and hence the subgraphs that implement their function can be included in the program graph. However, in object-oriented languages, it is often not possible to know even the type of object in use. In Smalltalk, we know only that the object supports some method. Exactly how the method is implemented, i.e., its subgraph, cannot be determined until run-time. Thus, we must be able to delay the binding of subgraphs for method implementation until run-time. Providing dynamic graph elaboration of actors into subgraphs would satisfy this need. As an example consider the C++ code fragment in Figure 2. We cannot know at compile time whether to use the program subgraph for

```
class shape {
public:
   virtual void draw();
}
class circle: public shape {
public:
   virtual void draw();
}

class square:public shape {
public:
   virtual void draw();
};

void draw_screen() {
shape *shape_ptr;
for each shape in shape_list
   shape_ptr->draw();
}
```

Figure 2. A static graph for function `draw_screen()` is inadequate because we do not know the types of the shapes.

the class `circle`, the class `square`, or some other class derived from `shape`. We could, of course, generate a subgraph that contained all possible graphs, and dynamically select the appropriate subgraph using control actors. This approach is undesirable since program graphs can become quite large, and, as we will see shortly, the use of control actors will significantly reduce the granularity of computation.

A second reason we require dynamic topology program graphs is that performance suffers when the topology of the graph is static due to the reduction of the granularity of computation. To see why the granularity is reduced consider how control is performed in static graphs. Certain actors in the program graph, called control actors, determine at run-time the actual execution pattern for the graph, such as which subgraphs are executed and how many loop iterations or recursive calls are invoked. Examples of control actors include greater than, less than, equal to, switches, and merges. If static graphs are used, we must use control actors to determine the actual execution pattern. Control actors have very small granularity, often a single instruction. There can be many control actors even for simple operations such as unrolling a loop or performing conditional and case statements. Studies have shown that there is a control flow statement (if, for, while) every 3-7 statements in typical programs. Further, we must pay the communication and scheduling overhead for each control actor executed just as we do for larger grain actors. To illustrate the problem with control actors consider the code fragment and program graph shown in Figure 3. In Figure 3 the function

```
float f(int dim, float initial){
  int i;float last=input;
  for (i=0;i<dim;i++)
     if (F1(dim,i,input))
        last = SG1(last);
     else
        last = SG2(last);
  return(last);
```

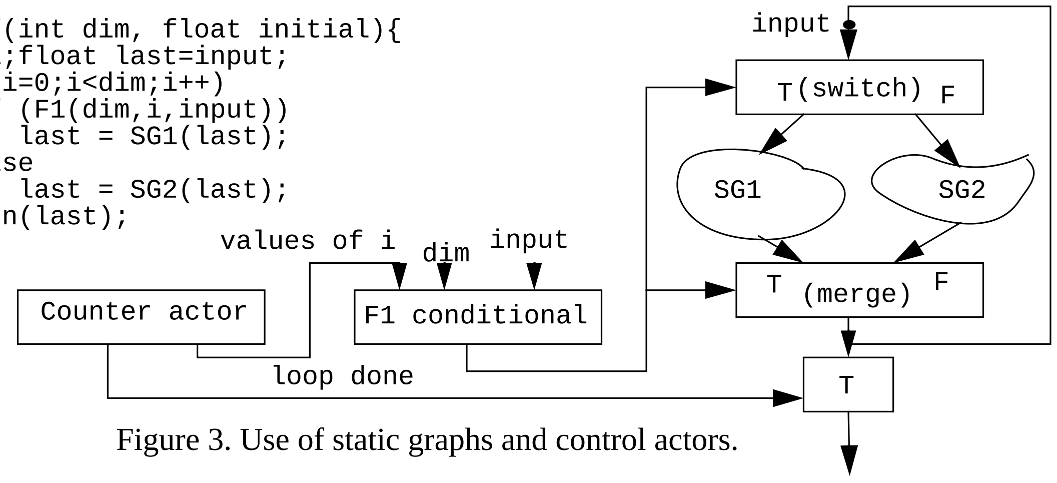


Figure 3. Use of static graphs and control actors.

`f()` performs `dim` iterations of a loop. In each iteration the function `F1(dim,i)`, based solely on local state information, determines whether `SG1` or `SG2` is executed. `SG1` and `SG2` correspond to statements that themselves form a subgraph and may contain additional control constructs. Suppose that `F1` is a simple function, then the effective granularity of this graph, ignoring the subgraphs `SG1` and `SG2`, is a few instructions at best. For distributed memory machines a far larger granularity is required.

Much of the control overhead can be eliminated by moving the control of subgraph selection and unrolling into actors and having the actors construct subgraphs at run-time that reflect the effects of performing the control function internally. By dynamically creating program subgraphs as the loop is unrolled we can substitute the overhead of firing small actors with the overhead of dynamically creating subgraphs.

The shortcomings of traditional data-flow led to the development of the macro data-flow model. The macro data-flow model is an extended data-flow model that addresses the problems with traditional data-flow. There are three principal differences with data-flow. First, as in other large-grain data-flow systems[4][6][9] the granularity of the actors is considerably larger; we call these *macro actors*. This provides the flexibility to choose an appropriate degree of parallelism. Second, some actors can maintain state information between firings, providing an effective way to model side-effects and non-determinism; we call these *persistent* actors. Third, unlink other large grain data flow systems the structure of macro data-flow program graphs is not fixed. Graphs grow by the run-time elaboration of graph nodes into arbitrary subgraphs.

Macro actors are large-grain actors that perform high-level functions such as matrix multiplication, Gaussian elimination or image convolution instead of individual machine instructions. The important characteristic of macro actors is that they are sufficiently computationally intensive to amortize the overhead costs. How the computation grains are specified is not a part of the model; that is a language issue. There are two types of macro actors, *regular* actors and *persistent* actors.

Regular actors are similar to actors in the data-flow model. Specifically, all regular actors of a given type are functionally equivalent. A regular actor is enabled and may execute when all of its input tokens are available. It performs some computation, generating output tokens that depend only on its input tokens. It may maintain internal state information during the course of a single execution, but no state information is preserved from one execution to another; they are pure functions.

Persistent actors maintain state information that is preserved from one execution to the next. Output tokens generated by a persistent actor during different executions are not necessarily the same for the same input tokens. The state corresponds to member variables (instance variables) in the object-oriented paradigm. This correspondence to member variables implies that several different actors may share the same state, as with the enqueue and dequeue operations on a queue. The model guarantees that the actors that share state will be executed in mutual exclusion, that is, no two actors that share the same state will ever be executing simultaneously. (This can be modeled in stateless data-flow using a single "state" token and a non-deterministic merge operator[1]. Only one actor can "possess" the state token at a time, guaranteeing mutual exclusion.) Thus, the set of actors that share state combined with the state they share can be thought of as a monitor.

The introduction of state means that the arcs of the program graph no longer model all dependencies in the program; there are implicit dependencies via the shared state. For example, consider the program graph fragment in Figure 4. Suppose that actors A and B share state. If the execution of A occurs first, there is a hidden dependency, based on the state, between A and B. Because of this hidden dependency the results of the A and B operations depend not only on their arguments and the object history, but also on the order of execution.

The introduction of state has one very important consequence: in macro data-flow some programs will be determinate, and others not. We have found that for many scientific codes, the programs are determinate because of the way the actors are used. In general though, this is not the case.

Non-determinism is not necessarily bad. There are in fact many "correct" non-deterministic applications. Concurrent database applications in particular are non-determinate. They do have a notion of correctness called consistency [7] based on the idea of transaction serializability. However, the enforcement of monitor properties alone on objects, as is done in MDF, is not sufficient to guarantee serializability. Thus, in macro data-flow, as in monitor systems in general, it is the responsibility of the programmer to guarantee higher-level notions of correctness.

3.2 Dynamic Graph Representation Using Futures

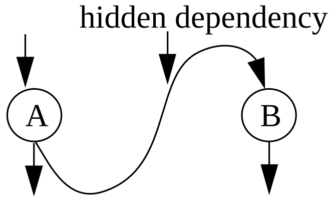


Figure 4. Hidden dependencies develop when A & B share state and A is executed before B.

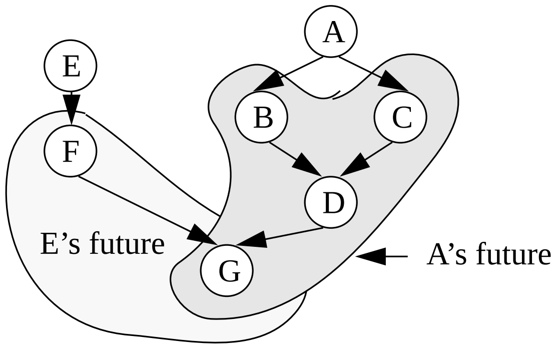


Figure 5. A macro data flow subgraph. The future of the actor A is shown.

Some mechanism is required to represent program graphs in any graph-based model of computation. The realization of a mechanism often involves a trade-off between different design objectives, including performance. The design of the graph representation mechanism used in the MDF model is driven by the theoretical requirements of the model and by implementation and performance considerations. The model requires that the mechanism must support dynamic program graphs, and both persistent and regular actors. Implementation and performance considerations require 1) that the mechanism use a distributed, as opposed to centralized, control, 2) that graph updates be a non-global, local operation, 3) that no shared memory be assumed, and hence messages must carry flattened graph segments, and 4) that actors are context independent (they do not name their input/output communication partners).

Program graphs are represented in MDF using *futures*.[3] 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 5. *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.

In many respects a future is like a parallel form of a continuation[1]. Both represent the remainder of the computation at a particular instant. The difference between a future and a continuation is that a future does not encapsulate an environment, and can only be used once, while a contin-

3. MDF futures should not be confused with Multilisp futures[20]. Multilisp futures represent a promise to deliver a value in the future. MDF futures represent the future of a computation, and have more in common with continuations[1].

uation can be used again. In addition, continuations are sequential, while futures may be parallel.

When an actor such as *A* receives its tokens, it receives with them a future. 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. *A*'s future is broken into two futures. These futures, along with the value 5, are forwarded to *B* and *C*. The new state is shown in Figure 6. *B* and *C* are now enabled and may execute since they have a token (value) on each input arc.

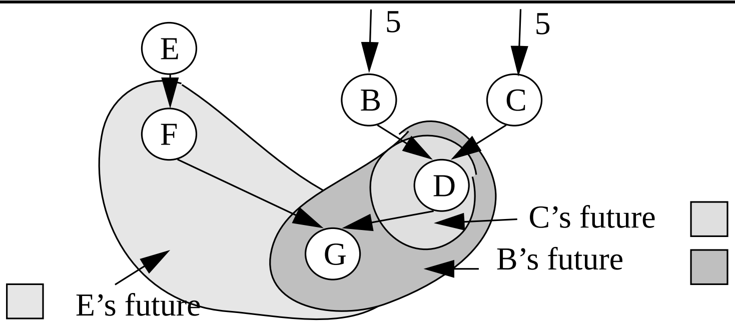
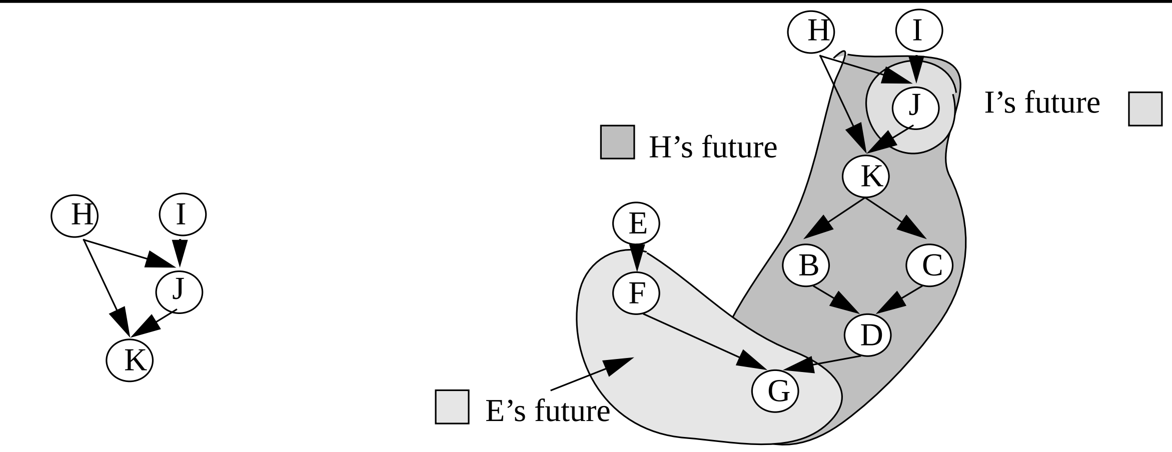


Figure 6. New futures after actor A from Figure 5 returns a value.

Alternatively, *A* may elaborate itself into an arbitrary subgraph. Suppose that *A* elaborates into the subgraph shown in Figure 7(a). The new state is shown in Figure 7(b). In this case the graph has grown, rather than contracted as in Figure 6. The key point is that in both cases only *A*'s future needs modification. Neither *E*, nor any other actor or system component need be notified of the change.

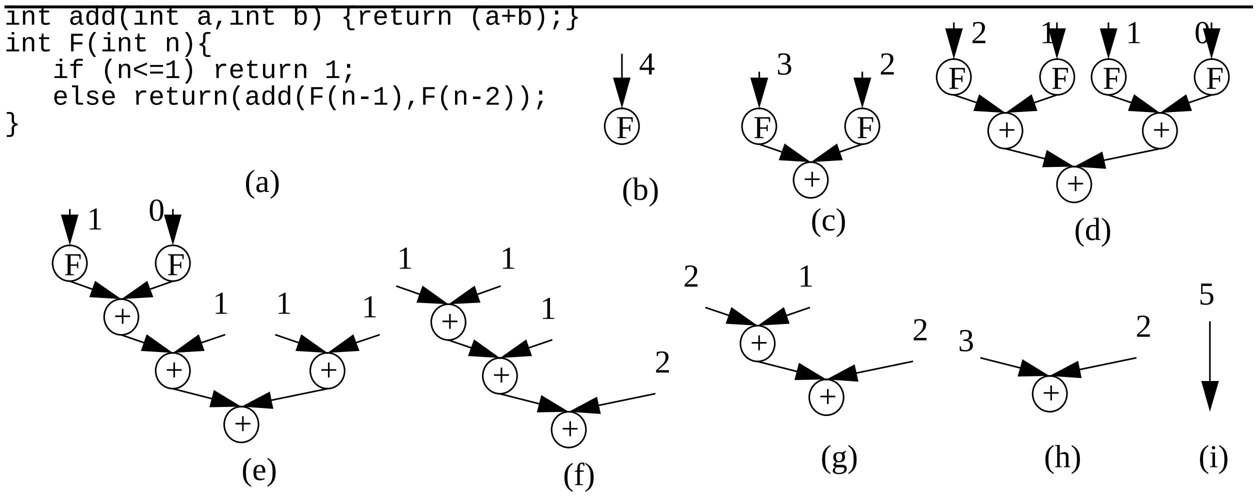
Example 1 - Fibonacci Numbers

Consider the naive implementation of finding a fibonacci number given in Figure 8. Assume two types of actors, an integer adder and a fibonacci function `int F(int)`. Suppose an *F* actor is invoked with a token value of 4 (the mapping of code fragments to MDF will be discussed later). The initial graph marking is shown in Figure 8 (b). Because the input value is greater than 1, the actor *F* elaborates itself into the graph shown in 8 (c). The successive elaborations are shown in 8 (d) though



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

Figure 7. Actor elaboration. The actor A from Figure 5 elaborates into the subgraph of (a). The resulting graph is shown in (b).



```
int add(int a,int b) {return (a+b);}
int F(int n){
   if (n<=1) return 1;
   else return(add(F(n-1),F(n-2));
}
```

Figure 8. Successive elaborations of the fibonacci function for F(4).

8 (i). Note that the order shown is just one possible ordering.

An important consequence of the use of futures as a graph description mechanism is that graph control is completely decentralized. Each actor receives enough of the program graph to continue the computation. There is no need to coordinate the execution of separate subgraphs, and their execution may proceed independently. Furthermore, it is not necessary for the entire program graph to be generated at compile time. Indeed, the structure of the graph is only implied, and changes as actors modify their futures. As an example of decentralized control, recall the subgraph shown in Figure 6. Suppose that *E* is executing on processor `P1` and that *A* is executing on processor

`P2`. Now suppose that *A* elaborates itself into a subgraph as in Figure 7. No communication between `P2` and `P1`, or any other processor, is necessary. The future of *A* can be elaborated locally without any external notification. Decentralized control is important in order to satisfy the bounded resources principle: "The service demanded from any component *(of the system)* should be bounded by a constant. The constant should be independent of the number of nodes."[28] If the service demanded is not bound by such a constant, then that component will become a bottleneck and limit the scalability of the system.

Another consequence of the use of futures is that actors are context-independent. Actors need not be aware of the source of their arguments or the destination of their results. For example, in Figures 6 and 7 the actors *B* and *C* cannot distinguish between the two graphs, or between graphs that have different numbers of outgoing edges. The use of context-independent actors contrasts sharply with CSP-based systems[21][22], where each computational unit knows the names of its communication partners. Context-independence is important for software reuse. Without it, modules must be tailored by hand for each context in which they may be used.

4. The Mentat Programming Language (MPL)

Rather than invent a new language for writing parallel software, MPL is an extension of the object-oriented language C++. The extensions are designed to facilitate communication about granularity and data decomposition between the programmer and the compiler and run-time system. The extensions are how we encode programmer knowledge so that the compiler and run-time system (RTS) can make better decisions. The extensions are applicable to a range of imperative languages.

In MPL we have extended the object-oriented concepts of data and method encapsulation to include *parallelism encapsulation*. Parallelism encapsulation takes two forms that we call *intra-object* encapsulation and *inter-object* encapsulation. Intra-object encapsulation of parallelism means that callers of a Mentat object member function are unaware of whether the implementation of the member function is sequential or is parallel, i.e., whether its program graph is a single node, or whether it is a parallel graph. Inter-object encapsulation of parallelism means that programmers of code fragments (e.g., a Mentat object member function) need not concern themselves with the parallel execution opportunities between the different Mentat object member functions they invoke.

```
1  mentat class bar {
2  // private member functions and variables
3  public:
4     int function1(int);
5     int function2(int, int);
6  };
```
Figure 9. A Mentat class definition. Without the keyword “mentat” it is a legitimate C++ class definition

Instead this is managed by the compiler working in conjunction with the run-time system.

The basic approach in MPL is to allow the programmer to specify those C++ classes that are of sufficient computational complexity to warrant parallel execution. This is accomplished using the keyword `mentat` in the class definition. Instances of Mentat classes are called *Mentat objects*. The programmer uses instances of Mentat classes like any other C++ class instance. The compiler generates code to dynamically construct and execute macro data-flow graphs at run-time in which 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 is 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 data-flow subgraph. That is called intra-object parallelism encapsulation; the caller only sees the member function invocation.

There are four MPL extensions to C++: Mentat classes (both persistent and regular), the Mentat class member functions `create`() and `destroy`(), the `mselect`/`maccept` guarded statements, and the `rtf`() (return to future) function[19]. We will limit our discussion here to the most important of these with respect to parallelism, Mentat classes and `rtf()`.

4.1 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 9. The keyword mentat indicates to the compiler that the member functions of the class are computationally expensive enough to be worth doing in parallel. Member functions of Mentat classes correspond to actors in MDF. Mentat classes are further 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.

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.

Instances of Mentat classes are Mentat objects. 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. Parameter passing is by *value*. 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 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.

4.2 Member Function Invocation

Member function invocation on Mentat objects is syntactically the same as for C++ objects. Semantically there are three important differences. First, Mentat member function invocations are non-blocking, providing for the 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 does not assume shared 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. The effect is that the memory object to which the pointer points is copied. Variable size arguments are supported as well, as they facilitate the writing of library classes such as matrix algebra classes.

4.3 The Return-to-Future Mechanism

Mentat member functions use the `rtf()` as the mechanism for returning 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.* If the caller does not use the value, a copy is not returned.

While there are many similarities between the C `return` and `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 saves on communication overhead. Second, a C `return` signifies the end of the computation in a function, while an `rtf`() does not. An `rtf`() indicates only that the result is available. Since each Mentat object has its own thread of control, additional computation may be performed after the `rtf`(), e.g., to update state information or to communicate with other objects. By making the result available as soon as possible, we permit data dependent computations to proceed concurrently with the local computation that follows the `rtf`(). 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. Indeed, the result may be computed by a parallel subgraph obtained by detecting inter-object parallelism.

5. The MPL Compiler

Given that the user has specified the computation boundaries and granularity via Mentat class specification, the remaining problem is to manage communication and synchronization by mapping the application to macro data-flow graphs. The MPL compiler and the run-time system perform this task.

As shown in Figure 10, the MPL compiler (mplc) takes MPL programs as input and generates C++ code as output. The C++ code includes library function calls that interact with the run-time

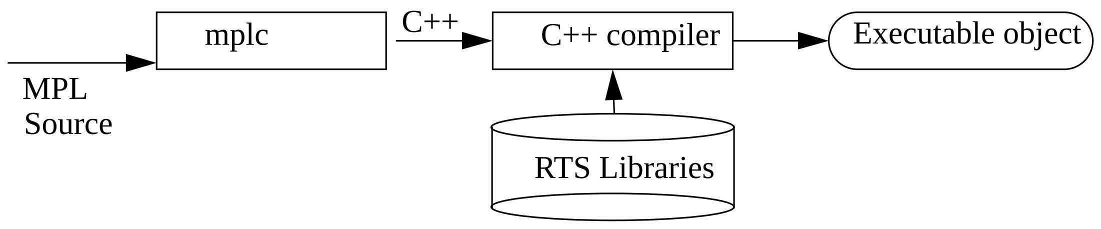


Figure 10. MPL compiler steps.

system to perform *run-time* graph construction, communication, and synchronization. As in any compiler there are many interesting issues. We will restrict ourselves to those aspects of code translation relevant to data-flow detection.

5.1 The Compiler's View of the Run-Time System

The basic compilation problem is to map MPL constructs onto the MDF model and its realization in the run-time system (RTS). The RTS supports an object model in which each Mentat object instance corresponds to a process. Mentat class member functions correspond to MDF actors. 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.

The data dependencies between Mentat object function invocations correspond to arcs in the MDF program graph. Actor elaboration corresponds to intra-object parallelism encapsulation. This occurs when a Mentat object member function uses other Mentat object member functions such that a subgraph is generated, and the member function performs an `rtf(variable)` where `variable` is a value that will be generated by the subgraph. A form of intra-object parallelism also occurs when an actor elaborates but does not return the subgraph.

The RTS provides functions that instantiate new object instances, and perform token matching, run-time data-flow detection, run-time program graph construction, and actor elaboration. A complete discussion of the RTS is given in [17]. We consider here three sets of services that the RTS provides, Mentat object back-end processes, `mentat_object` front-end classes, and the data-flow detection library.

The RTS implementation of Mentat objects consists of two components, (1) a front-end class, `mentat_object`, that contains the name of a Mentat object (process) and is the handle by which the back-end object is manipulated, and (2) a back-end server object process that contains the Mentat object's state and performs the member functions. 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 11.

```
class mentat_object {
   object_name i_name;
public:
   CIP invoke_fn(int,int,...); // used to communicate with back-ends
   void create(); // instantiate new back-end
   void destroy();// destroy the back-end
};
```

Figure 11. Partial interface of the front-end `mentat_object` class.

The data-flow detection library consists of routines that 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 12) 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.

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

(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.

Figure 12. Two uses of result variables. In this example, bar is a regular Mentat class.

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);.
```

We keep track of Mentat object member function invocations at run-time using c*omputation instances*. A computation instance corresponds to a node in a MDF program graph. It contains the name of the Mentat object invoked, the number of the invoked 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. A computation instance contains sufficient information to acquire the value that is the result of the operation.

The `mentat_object` member function `invoke_fn()` is called when a Mentat object member function is used. It creates a new computation instance for the computation, (i.e., a program graph node is created), and marshals both actual arguments (e.g., integers) and arguments that are computation instances. 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 12. 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 data-flow 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 data-flow 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. 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`:

```
1)  SET_ME((char*) rv_address, CIP node);
2)  RV_DELETE((char*) rv_address);
3)  force();
4)  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. `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. This is the mechanism for removing a PRV from RVS.

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

The 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. Once the result is available, `RESOLVE`() places the result into the memory to which the `rv_address` points.

5.2 Translation

The compiler is a four pass compiler. In the first pass, the MPL source is parsed and the parse tree and symbol tables are constructed. Mentat classes are identified and marked in the symbol table. Front-end classes are constructed by derivation from the class `mentat_object`. Function numbers are assigned to all Mentat class member functions. The function numbers are used later as arguments to `invoke_fn()`.

The second pass marks all variables that are PRVs. We do this by descending the parse tree and checking if the right-hand side of an assignment is a member function invocation on an object whose class is a Mentat class. If so, the left-hand side is marked as a PRV. In the simple case, the left-hand side is a simple variable `w` as in Figure 12. In the case where the left-hand side is more complex, such as `Z[i]` or `Y.X`, the outermost enclosing variable, e.g., `Z` and `Y`, are marked as PRVs.

The third pass traverses the parse tree again, transforming the tree. The transformations result in the generation of code to call the RTS library routines that perform the desired action at run-time. There are five transformations:

1) When a PRV occurs on the left-hand side of a Mentat expression, add it to RVS using `SET_ME()`, and construct a computation instance (add it to the graph) using `invoke_fn()` on the front-end.

2) When a PRV occurs on the right-hand side in a strict expression, `RESOLVE()` it.

3) When a PRV occurs on the left-hand side of a non-Mentat expression, or goes out of scope, remove it from RVS using `RV_DELETE()`.

4) When a PRV occurs as an argument to a Mentat expression, add an arc from the computation instance corresponding to the PRV to the computation instance corresponding to the Mentat expression. The compiler has detected a potential data dependency. This is done using `invoke_fn()` and passing in the address of the PRV.

5) Marshall all non-RVS arguments to Mentat invocations directly, i.e., package them into messages to send to the object using `invoke_fn()`.

The fourth and final pass traverses the transformed parse tree and prints it out. If the source being compiled is the code for a Mentat class back-end (as opposed to a main program) the compiler also generates a server loop member function and a `main`() function. The server loop is equivalent to a large select/accept in which every member function of the class is included. The output from the fourth pass can be used directly as input to a C++ compiler.

5.3 Translation Examples

The following two examples illustrate the code translation process, and the parallelism that results from run-time elaboration of the program into MDF graphs. For each example we present the MPL code, the translated code, and the resulting program subgraph. The original code is retained in comments. Only statements involving PRV's and Mentat objects are transformed.

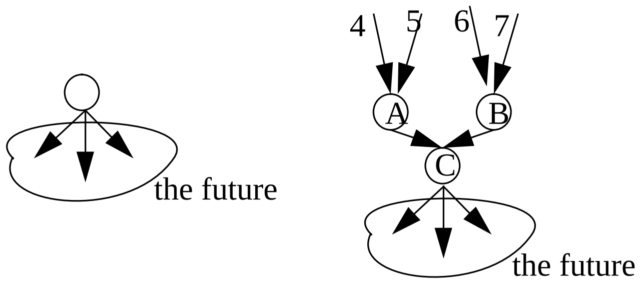
Example 2: Simple Mentat object invocation. In Figure 12 two program fragments were presented to illustrate blocking versus non-blocking member function invocation. The MPL translations are shown in Figure 13 (a) and 13 (b) respectively. The code fragment of Figure 12 (a) has

```
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));
```

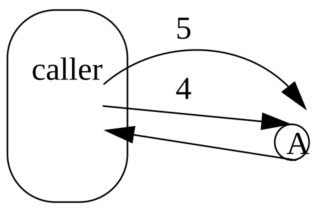
(a) Code transformation for 12 (a)

```
bar A;
int w,x,y;
//w = A.op1(4,5);
(*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 12 (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 13. Code transformations and generated graphs for code fragments of Figure 12.

been extended to include an `rtf(y)`. The result is that the actor containing the code fragment is elaborated into the subgraph shown.

Example 3: Pipelined Functional Parallelism. Or final example is the most complete translation. It illustrates the dynamic construction of program graphs and the use of regular Mentat classes. We define four classes, two variable-sized C++ classes, `string` and `dblock`, and two Mentat classes, `regular mentat class data_filter`, and `persistent mentat class mfile`. The `mfile` class provides operations to open a file, read blocks, and write blocks of data. The `data_filter` class provides two different filter operations, `filter1`() and `filter2`(). What the filters do is unimportant. It is important that they are pure functions that depend only on their inputs. The main loop reads data blocks from the input file, passes them through two filters, and writes the results to the output file (Figure 14). The effect of executing the code fragment is to generate MAX_BLOCKS copies of the program graph of Figure 15. Taken together, these graphs form an execution pipe.

Several observations can be made from this example. First, since the variable `dp` is a regular Mentat class, the system is free to instantiate new instances at will. If the filter operations are computationally heavy relative to the reads, there will be many concurrent instances. Second, the main loop may have executed to completion (all `MAX_BLOCKS` iterations) before the first write has completed. Third, suppose our “caller” (the main loop) was itself a server servicing requests for clients. Once the main loop is complete the caller may begin servicing other requests while the first request is still being completed. Fourth, the order of execution of the different stages of the different iterations can vary from a straight sequential ordering. This can happen, for example, if the different iterations require different amounts of filter processing. This additional asynchrony is possible because the run-time system guarantees that all parameters for all invocations are correctly matched, and that member functions receive the correct arguments. The additional asynchrony permits additional concurrency in those cases where execution in strict order would prevent later iterations from executing even when all of their synchronization and data criteria have been met. Finally, in addition to the automatic detection of inter-object parallelism opportunities, we may also have intra-object parallelism encapsulation, where each of the invoked mem-

```
persistent mentat class mfile {
   // ... locals
public:
   int open(string*);
   void write(int offset;int bytes;dblock *data);
   dblock *read(int blk_num);
};
regular mentat class data_filter {
public:
   dblock* filter1(dblock*);
   dblock* filter2(dblock*);
};

// Now a code fragment that uses the above definitions
1: mfile in_file,out_file;
2: data_filter dp;
3: dblock *res;
4: in_file.create(); // Create a persistent mfile.
5: out_file.create(); // Create a persistent mfile.
6: x= in_file.open(”infile”); // non-blocking call
7: y= out_file.open(”outfile”);// non-blocking call
8: if ((x < 0)|| (y<0)) {/*handle the error*/} // Note strict on x & y
9: for (i=0,i<MAX_BLOCKS;i++){
10:   res=in_file.read_block(i); // non-blocking
11:   res = dp.operation1(res); // arc constructed, new instance
12:   res = dp.operation2(res); // arc constructed, new instance
13:   out_file.write_block(i,res); // non-blocking, arc constructed
   }
-------------------- translation ---------------------
// Note front-end derivation off of mentat_object
class mfile : public mentat_object {
   public:
      mfile(){set_object_name(&i_name, “mfile”, _M_PERSISTENT);}
}
class data_filter : public mentat_object {
   public:
      data_filter(){set_object_name(&i_name, “data_filter”, _M_REGULAR);}
}
:  mfile in_file,out_file;
2: data_filter dp;
3: dblock *res;
4: in_file.create(); // Create a persistent mfile.
5: out_file.create(); // Create a persistent mfile.
// x= in_file.open(”infile”); // non-blocking call
6:(*SET_ME((&x)))=in_file.invoke_fn(101,1,STRING_TO_ARG(“infile”));
// y= out_file.open((string*)”outfile”);// non-blocking call
7:(*SET_ME((&x)))=in_file.invoke_fn(101,1,STRING_TO_ARG(“infile”));
8: if (((RESOLVE(&x),x) < 0)|| ((RESOLVE(&y),y)<0)) {}
// if ((x < 0)|| (y<0)) {/*handle the error*/} // Note strict on x & y
9: for (i=0,i<MAX_BLOCKS;i++){
//    res=in_file.read_block(i);
10:   (*SET_ME((&res)))=in_file.invoke_fn(102,1,VAR_TO_ARG(&i,4,0));
//    res = dp.operation1(res); // arc constructed, new instance
      (*SET_ME((&res)))=dp.invoke_fn(101,1,PRV_TO_ARG(&res,4,0));
//    res = dp.operation2(res); // arc constructed, new instance
12:   (*SET_ME((&res)))=dp.invoke_fn(102,1,PRV_TO_ARG(&res,4,0));
//    out_file.write_block(i,res); // non-blocking, arc constructed
13:   (*SET_ME((&res)))=out_file.invoke_fn(103,2,VAR_TO_ARG(&i,4,0),
            PRV_TO_ARG(&res,4,0));
}
// Note: PRV_TO_ARG and VAR_TO_ARG are argument marshaling functions that are
// called on PRV’s and non-PRV’s respectively.
```

Figure 14. Program and translation illustrating pipeline parallelism.

ber functions, e.g., the filter operations, may be internally parallel.

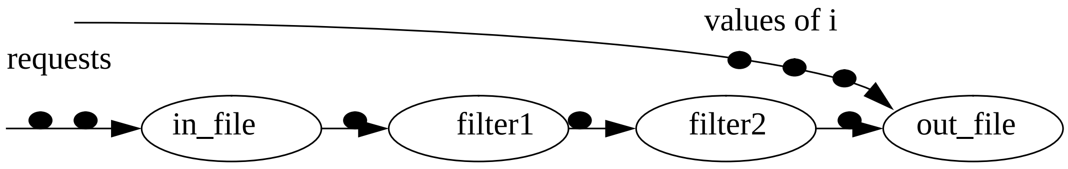


Figure 15. Subgraph of loop body of Figure 14. Multiple invocations result in pipelined execution.

6. Summary

The difficulties associated with explicitly managing medium-grain parallelism combined with the inability of compilers to automatically extract medium-grain parallelism have plagued applications writers for years. In this paper we presented the macro data-flow model and a technique for automatically extracting medium-grain parallelism from programs once the programmer has identified the computation grains using object class definitions. The macro data-flow model is a parallel, data driven computation model that is scalable, medium grain, supports the object-oriented paradigm, and can be efficiently implemented on distributed memory MIMD machines. Our hybrid approach to finding and exploiting parallelism synergistically combines the advantages of both explicit and compiler-based techniques. We capitalize on both the programmer’s domain knowledge and the compiler’s ability to safely manage communication and synchronization.

The bottom line for any parallel computation system is application performance, in particular, speedup relative to a sequential implementation of the application. We have extensive experience with Mentat performance on applications from areas as diverse as electrical engineering, physics, biochemistry, and computer science, on platforms as diverse as networks of workstations and the Intel iPSC/860 (gamma). The results are detailed elsewhere [15][16]. In several cases hand-coded parallel implementations of the application exist. These provide us with a metric to measure the penalty of using Mentat and MDF. The results are very encouraging. Performance is good, and competitive with the hand-coded implementations. Further, the use of the object-oriented paradigm combined with the compilation techniques used has reduced development time, and more importantly, made modifications to the applications easier.

7. References

[1] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge Massachusetts, 1985.

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

[3] J. R. Allen, and K Kennedy, “PFC: A Program to Convert FORTRAN to Parallel Form,” *Proceedings of the IBM Conference on Parallel Computers and Scientific Computations*, Rome, 1982.

[4] G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings Publishing Co., Redwood City, CA., 1989.

[5] Arvind and J. D. Brock, “Resource Managers in Functional Programming,” *Journal of Parallel and Distributed Computing*, vol.1, pp. 5-21, 1984.

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

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

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

[9] P. A. Bernstein, and N. Goodman, “Concurrency Control in Distributed Database Systems,” *ACM Computer Surveys*, pp. 185-221, vol. 13:2, June, 1981.J.

[10] Boyle et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, New York, 1987.

[11] 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.

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

[13] N. Carriero, and D. Gelernter, “How to Write Parallel Programs: A Guide to the Perplexed,” *ACM Computing Surveys*, pp. 91-125, vol. 23, num. 1, March. 1991.

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

[15] G. Fox et al.,*Solving Problems on Concurrent Processors Volume I*, Prentice Hall, Englewood Cliffs, NJ, 1988.

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

[17] A. S. Grimshaw, E. A. West, and W.R. Pearson, “No Pain and Gain! - Experiences with Mentat on Biological Application,” to appear in *Concurrency: Practice & Experience*, Vol. 5, issue 4, July, 1993.

[18] A. S. Grimshaw, W. T. Strayer, and P. Narayan, “Dynamic Object-Oriented Parallel Processing,” to appear, *IEEE Parallel & Distributed Technology: Systems & Applications*, May, 1993.

[19] A. S. Grimshaw. The Mentat Run-Time System: Support for Medium Grain Parallel Com-

putation. *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 1064-1073. Charleston, SC., April, 1990.

[20] 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.

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

[22] 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.

[23] C.A.R. Hoare, “Communicating Sequential Processes,” *Communications of the ACM*, pp. 666-677, vol. 21, no. 8, August, 1978.

[24] Inmos Ltd., Occam Programming Manual, *Prentice-Hall*, New York, 1984.

[25] Intel Corporation, “iPSC/2 USER’S GUIDE”, Intel Scientific Computers, Beaverton, OR, March 1988.

[26] D. Kuck, R. Kuhn, B. Leasure, D. Padua, and M. Wolfe, “Dependence Graphs and Compiler Optimizations,” *ACM Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp. 207-218, January, 1981.

[27] T. G. Lewis and H. El-Rewini, Introduction to Parallel Computing, Prentice Hall, Englewood Cliffs, NJ, 1992.

[28] S. Mullender ed., *Distributed Systems*, ACM Press, 1989.

[29] A. Osterhaug “GUIDE TO PARALLEL PROGRAMMING On Sequent Computer Systems,” *Sequent Technical Publications*, Sequent Computer Systems, Beaverton, OR, 1989.

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

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

[32] M. J. Quinn, *Designing Efficient Algorithms For Parallel Computers*, McGraw-Hill Book Company, New York, 1987.

[33] V.S. Sunderam, “PVM: A framework for parallel distributed computing,” *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, December, 1990.

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