Sample, Neal J., MARS: Multidisciplinary Application Runtime System, M.S.,

Department of Computer Science, July 1998.

Expert knowledge from many disciplines is frequently embodied in stand-alone codes used to solve particular problems. Codes from various disciplines can be composed into cooperative ensembles that can answer questions larger than any solitary code can. These multi-code compositions are called *multidisciplinary applications* and are a growing area of research. To support the integration of existing codes into multidisciplinary applications, we have constructed the Multidisciplinary Application Runtime System (MARS). MARS supports legacy modules, heterogeneous execution environments, conditional execution flows, dynamic module invocation and realignment, runtime binding of output data paths, and a simple specification language to script module actions.

MARS: A Multidisciplinary Application Runtime System

by
Neal Joseph Sample

A Thesis Submitted to the
Department of Computer Science and
The Graduate School of the University of Wyoming
in Partial Fulfillment of Requirements
for the Degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Laramie, Wyoming
July, 1998

This thesis is dedicated to my parents, Kenneth and Sharon Sample. They instilled in me the strong ethic that has carried me thus far, and continues to carry me into the future. Thank you both.

Acknowledgements

There are too many people to thank without a second volume as large as this one. However, I would like to single out the following individuals who have contributed that much more during my stay here at the University of Wyoming (*A-Z order*).

Mark Arnold -- For the long talks (and low rent!). You were always there to listen and discuss the forks not taken, in CS and life.

Carl Bartlett -- For the help. For the headaches and breakthroughs. For the distractions. For the laughs. You never were an assistant, as much as a friend and ally.

Matthew Haines -- For your support these past two years, for your excellent advice, and a boost over the hurdles.

Duncan Harris -- For the many doors you opened for me (though I didn't always cross the thresholds). I appreciate your efforts to help me help myself.

Meredith Marine -- For putting up with me for so many years, with so many laughs, through so many tears. For being Tiny Elvis.

Chris Rothfuss -- For the Deutschermeister and the *friction*. For being on my side, on the various fields of conflict, and off.

Robert Torry -- For your direction and sage advice during my early years. A true master of the dialectic, you lead me to answers I didn't realize that I already had inside.

I *acknowledge* the following people for being particularly groovy (as is my right!): **Tom Bailey**, **Chang-Yul Cha**, **Mark Dixon**, **Neil Harrison**, and **Peter Shive**.

Table of Contents

List Of Figures

I. Problem

As expert knowledge is more and more frequently embodied in computer models and simulations, the opportunity to compose useful models from multiple disciplines is increasing as well. In order to better solve many complex problems, it is useful to draw together domain expertise from multiple disciplines.

For instance, designing an automobile requires information about structure, propulsion, guidance, safety, and many other features. Designs are done in teams, with each team member (or group) having specific tasks, completing one part of the puzzle. The group responsible for a new engine design is not the same group responsible for the body design. Further up the scale, the entire design team could be seen as one small component themselves, coupled with advertising, sales and manufacturing teams to deliver the product to market. The combination of these different expert groups can achieve much more than any group by itself.

The automobile production problem is analogous to what is occurring in scientific research today. Models are taken from different disciplines and combined in innovative ways to provide answers that the component models could not provide by themselves. The problem of combining models from more than one discipline is known as the *multidisciplinary optimization problem*, and the resulting applications are called multidisciplinary applications.

Significant recent work has been devoted to the problem of multidisciplinary applications (MDO applications). Multidisciplinary applications are collections composed of programs bound together to perform a single task. The individual component programs come from varied disciplines to solve a problem “larger” than the problem solved by any of the individual component. “Larger” may mean in terms of overall complexity, geographic scope, the ability to handle various data types (*applicability*), or in terms of other design factors. Multidisciplinary applications are used for global climate modeling, aircraft and automobile design, biological process modeling, and countless other tasks.

The interactions between components of an MDO vary for different problem types, and there are different methods being used to “glue” the components together. In most cases, the MDO is treated like a sequential pipeline, with information passed from one stage to the next, repeating the entire process as required. When a homogenous collection of modules is built from the ground up at a single site, the problem of passing information from one component to another is relatively easy; designers *choose* the interfaces, so the modules can cooperate out-of-the-box. This type of application is *could be* multidisciplinary, though not cost effective when reusable codes component codes are available.

Because available components used to build a single MDO are often heterogeneous in terms of component source language, data format requirements, and scope of functionality, they are often *forced* into a single execution format: sequential pipelines sharing data through files. The files for sharing data are frequently transformed between MDO components so that one component’s output is converted to an appropriate input for another component. This transformation is up to the MDO programmer, of course. So too are solutions to the problems of process creation and synchronization.

These can be done with script files, UNIX programs, or by using an additional control language.

To further illustrate the types of MDOs in actual use, we offer an example of a multidisciplinary code also introduced in several related references [Haines95a, Chapman94, ICASE95]. The example is of an MDO used for aircraft design. The aircraft design process draws from several different disciplines, including propulsion, aerodynamics, and structural analysis. Simpler component models from the various disciplines can be tied together into a collection that solves the larger problem of “aircraft design”. While our example is a simplification of the entire aircraft design process, it demonstrates the essential components of an MDO application. This example has three component modules, the *Optimizer*, the *FlowSolver* and the *FeSolver*.

The *Optimizer* invokes the other two tasks (*FlowSolver* and *FeSolver*) and generates an initial geometry. The *FeSolver* module generates a finite element model (to represent the aircraft) based on this geometry and uses some initial forces to determine the structural deflections. At the same time, the *FlowSolver* generates an aerodynamics grid based on the initial geometry and performs an analysis of the airflow around the aircraft, producing a new airflow solution. In subsequent runs, the *FeSolver* uses forces based on the current flow solution provided by the *FlowSolver* to produce new deformations. The *FlowSolver* uses the deformed geometry from the *FeSolver* and the previous flow solution to create new solutions. This process continues until the differences between current and previous *FeSolver* solutions are within some specified tolerance. After a solution is reached, the results are stored for offline use (Figure 1).

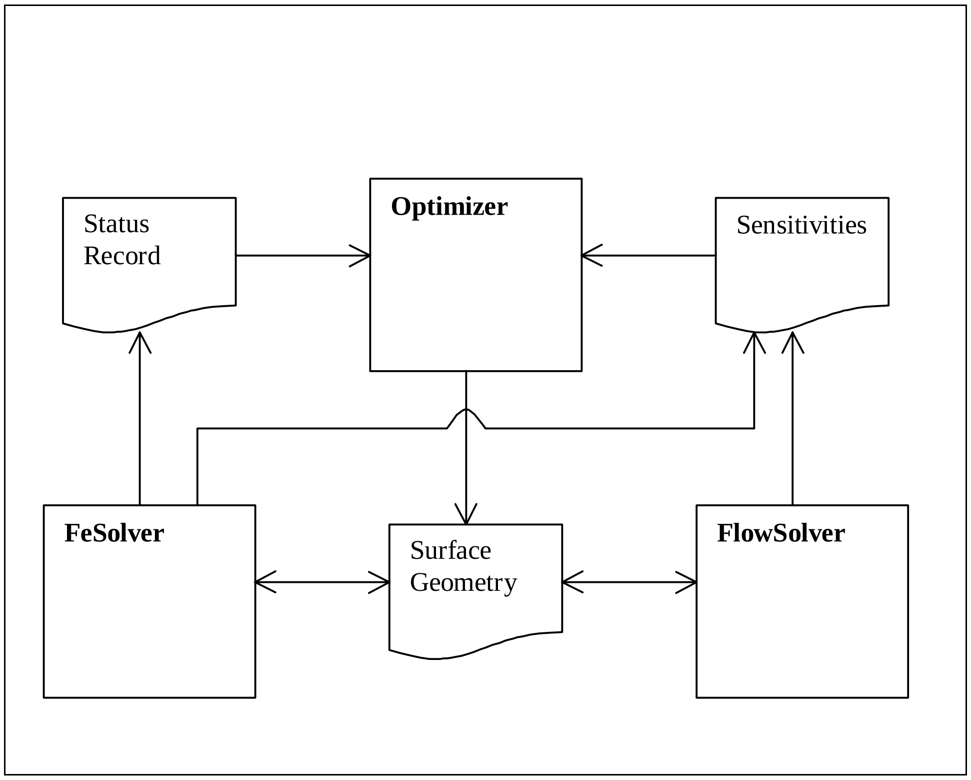

Figure 1. A sample MDO for aircraft design [Haines95a]

The aircraft design MDO has inherent task parallelism, though the writer of the *Optimizer* module must take care to properly synchronize the *FeSolver* and *FlowSolver*. A frequent characteristic of the current MDO application design process is that the MDO *builder*[1] is responsible for everything beyond the functionality of the particular working module.

The general MDO construction problem has three main components: managing concurrency and parallelism, data exchange, and basic control functionality (such as module invocation and termination). When using heterogeneous codes, little can be done

[1] *Programmer*, *designer*, *composer*, and *integrator* are also terms used to describe the person tying the different modules together.

to combine them (without rewriting the working modules to be compliant with the flavor of the month task and data parallel language system); there are very narrow design paths MDO builders usually follow. The path to managing concurrency problems is often to serialize the component processes, thus ensuring consistency at a cost of efficiency. The path for data exchange is commonly through UNIX files. Basic functions, like invocations, can be done with scripting languages or with system calls, such as the `exec()` in UNIX.

An important consideration in the development of MDO codes is that desirable component modules often exist, but were not written to be used in a cooperative environment. These *legacy codes*, while capable of performing a desirable task, are often limited in use because they are difficult to tie together with other legacy codes. Even MDOs built with newer modules designed with interfaces that are mindful of *specific* legacy codes are ineffective. Interfaces to the resulting new MDO collections are often as idiosyncratic as the legacy code that is the basis of the MDO; it is therefore difficult to expand or interchange other modules with the new MDO. For instance, increasing the variety of *Phillips* screws in a package does not make that package work better with a *flathead* screwdriver. In essence, to use legacy codes, designers must completely build around them, suffering the costs incurred when coupling heterogeneous modules, or rewrite them to comply with local modules, duplicating significant effort already expended on the legacy code's original design.

It was with these many considerations (data exchange, concurrency and parallelism, basic functionality of a component system, and support for legacy codes) in mind that we built the ***M****ultidisciplinary* ***A****pplication* ***R****untime* ***S****ystem* (MARS). MARS supports abstract data passing[2], asynchronous partially-ordered message passing, remote service requests, dynamic module invocation, and various primitives for module control. As such, it could serve as the runtime support for another task parallel high-level language specification, like Opus (which was designed for building MDO codes) [Bal96, Chapman94, Haines95a].

MARS also includes components beyond the runtime system proper. To fully support the construction of MDO codes, MARS understands a simple specification language that defines the *working modules* (or *tasks*) within an MDO. A collection of working modules and a *leader* (specification) to guide those modules are referred to as a MARS *ensemble*. (The specification language is discussed further in Chapter III.)

MARS also has a server component that resides on all execution nodes (CPUs) that participate in a running ensemble. This server, called an *mdo_server*, functions to invoke working modules on specific execution nodes at the request of another module. In MARS, working modules are generally only invoked by a leader or *sub-leader* module. The components of a leader (and how sub-leaders and working modules are added to an ensemble that follows the leader) are also part of the specification language.

To support legacy modules, MARS includes a source to source compiler called the *wrapper*. The wrapper takes source for a working module, wraps all the I/O routines with MARS calls, and augments the original code to act as persistent server capable of processing MARS messages. This wrapper is *not* dependent on user augmentation of

---

[2] "Abstract" in the sense the data is simply *written*. Applications are not aware that data goes to disk, to the network, or elsewhere. They simply *write* the data and the runtime system directs it to an appropriate receptacle.

original source, as with a full language specification like Opus. Without any annotation of the source by the MDO builder, the MARS wrapper replaces the legacy code's original entry point with MARS module server code, while preserving access to the original entry point for later use. As such, a module like the *FeSolver* will be invoked, wait until it receives an execution directive, run to completion (passing its data out of the module), and then wait idle until needed or terminated.

A powerful module composition tool is needed to bring tools from various disciplines together to form multidisciplinary applications. When legacy modules are developed, their design generally does not consider how the module could be used as part of a future collections of modules. Legacy codes are frequently designed to achieve a single task in the best possible way. As such, whatever local tools, methods, and protocols are available are used to build the best possible *stand-alone* module. This local construction process makes it difficult to integrate modules from different disciplines when they do not use common data formats, common communication protocols, or even common development languages. MARS is a composition tool designed to bring modules together from multiple disciplines to form multidisciplinary applications.

II. Related Work

There have been many different approaches to achieving task integration. Most approaches deal with homogenous tasks rather than integrating codes from multiple sources. The MDO integration problem has been tackled several different ways: on a language level by augmenting data parallel languages with task parallelism primitives and extending task parallel languages to handle multiple interfaces, and using a case-by-case approach to the problem with unique (one-time) solutions. We have found that similarities exist between elements of particular systems and certain MARS components, but that the collected functionality of MARS has not been realized.

We examine a few approaches to the MDO integration problem, with a focus on message and data transfer. Since MARS is a *runtime system*, primarily existing to plumb data and message connections between heterogeneous[3] codes, we focus on how these things are done in other systems. Most conspicuously absent in these previously developed systems is support for legacy applications.

We have found one system very similar to MARS, in development at Stanford University, known as CHAIMS (***C**ompiling **H**igh-level **A**ccess **I**nterfaces for **M**ulti-site **S**oftware*). The application of MARS to integrating MDO codes, when viewed as a simple programming model, is being realized by the CHAIMS project (but at a much higher level of abstraction than MARS). For the most part, CHAIMS operates above current parallel language specifications, while MARS operates below language

[3] Of course, MARS will work with modules that have similar (homogenous) interfaces as well, but that problem is not nearly as interesting.

specifications, yet there is still a strong analogue between the two systems. Before examining this similarity further, an examination of different language-level approaches is warranted.

Language similarities and differences

A primary difference between MARS and virtually all languages and language extensions is the support for legacy codes. Opus, for instance, can handle annotated HPF code and compile the source to the Chant runtime system [Mehrotra94, Haines94] (or even to the MARS runtime system, with some modification). On the other hand, a source code in C for a program that performs a Fast Fourier Transformation has no meaning to Opus. Even converting the C code to HPF would require the user to modify the resulting HPF to use Opus primitives and objects for communication and synchronization. With MARS, that same FFT program can be wrapped to become a working module without source modification, and is easily incorporated into a MARS ensemble. The purpose of language specifications is generally to build *new* codes, including *new* MDOs. They are designed with support for legacy modules in mind. Limited code reuse (using legacy modules) can be achieved with these language-level specifications, but only by reusing other codes originally written using a specific language specification.

It should be further noted that many language specifications could be compiled to MARS, although MARS currently lacks a native threading system. (The initial focus of the MARS project was to develop the message passing layer, the I/O routines, and the basic control and concurrency primitives. The assumption is that MARS will be augmented with an existing threads package.) Because MARS contains a very simple specification language, it is generally not as full-featured as more mature systems, but the runtime primitives can generally implement the missing higher level functions.

Fx

Developed at CMU, Fx is an extension of the Fortran language. In Fx, parallel sections of code, called *tasks*, are invoked by a program as subroutines. Tasks communicate with one another by sharing input and output arguments in their *parent* space. A task's parent is the scope that invoked the task. Data may be passed to or received from a task only when it is called or returns, respectively. As such, specification of data flows must be provided at compile time in the form of *input* and *output* directives. The data flows are therefore static and cannot be changed at runtime. This method does have the advantage of possible compile-time path optimizations. Also, the data flows are deterministic, and no runtime destination translations are required. The actual cost of runtime destination lookups are low, however.

While good for the design of new task parallel codes, Fx is not particularly suited for MDO applications [Bal96]. First, within a single invocation, the Fx language provides excellent support for communication and parallelism. However, between two distinct codes, both written in Fx, the MDO builder is once again responsible for handling communication. In effect, a *static* (where modules are never added or removed) MDO could be easily built as a single Fx program, but altering the ensemble would require re-coding and recompiling the entire Fx code (or else the user has to build a runtime system to provide the needed support[4]).

[4] Don't. Use MARS instead.

The Fx *input* and *output* directives radically limit the usefulness of tasks in a parallel environment [Bal96]. If tasks must communicate frequently, routines must be frequently split at synchronization points so the task can share its data. This of course makes programming the tasks difficult and greatly increases the overhead of task invocation.

With the MARS primitives for module invocation and the ability to simulate synchronous module data output, it would be a suitable runtime system for Fx to compile to. At the same time, allowing asynchronous data transmission and reception without having to break modules into component subtasks seems to be a distinct advantage of MARS. Also, for evolving MDO ensembles, MARS' weak data flow specification also seems to be advantage. (Recall that data is simply "written" in MARS modules, it can always be dynamically routed to various locations or modules, without *a priori* knowledge of the modules that will be part of the ensemble.)

Linda

Linda is a unique model for developing parallel programs. It provides communication through tuples that are placed into a shared virtual tuple-space. Active tasks may remove tuples from the tuple space at any time. With the tuple communication in place, message passing is a straightforward process. Linda extensions are available for many languages, including C, C++, Fortran, and Scheme [Carriero89].

For homogeneous parallel processes, Linda can be very effective. However, Linda is not particularly well-suited to multidisciplinary applications for several reasons. Control over resource allocation within the virtual tuple space is limited. Invocation and distribution of working modules as “live tuples” in the tuple space offers little control as well [Bal96, Carriero89].

Legacy codes not written in Linda typically use a distinctly different I/O and execution paradigm, and are thus rather difficult to port to use Linda primitives. In many ways, it would be easier for MDO builders to use the naïve approach of trapping all output to a file and then transforming that file into a usable format for another module than to achieve the same results in tuple space. Perhaps for experienced Linda programmers, this would not be the case, but with Linda there is the ultimate task of converting all I/O routines to Linda-friendly (read: *tuple space aware*) methods.

Orca

Orca is a task parallel language that supports process allocation in a manner similar to MARS. Like MARS module invocations, Orca tasks can be dynamically created and mapped to specific execution nodes. However, there is no explicit message passing in Orca. Instead, communication is handled by applying user defined Abstract Data Type (ADT) operations on *shared objects*. Orca collections are good for coarse-grained parallelism as operations on ADTs are always exclusive and indivisible [Bal96].

Unlike Opus, which is a specification language, Orca is self contained (in that has its own compiler and runtime system). In this regard, Orca does go significantly further than MARS. With MARS, it would be possible to provide runtime support to system like Orca, but more is possible. MARS includes primitives appropriate for simultaneous divisible access to ADTs. Once again, since Orca is a complete programming language, users are limited to programming in Orca. This does limit the possibility of using legacy codes, and it also prevents users from easily extending the Orca specification. Because

MARS concentrates on coordination rather than programming, it has a broader range of usefulness to legacy codes.

Additionally, because Orca is compiled and has its own runtime system, users may find it difficult to add to the Orca system. Since MARS is built in clearly defined, user-accessible layers, inserting functionality at the various levels is much simpler than with Orca. Finally, extending higher-level MARS constructs with source language (such a C) primitives is also possible, whereas there is no such distinction with Orca.

Opus

The Opus language specification was designed for multidisciplinary applications [Bal96]. Opus takes a data-centric approach to the MDO problem, introducing the *ShareD Abtraction* (SDA) for coordination and communication. SDAs act as data repositories, communication channels, and occasionally as compute nodes within Opus ensembles.

The Opus language extends HPF with its task parallel programming constructs [Chapman94, Mehrotra94]. There are several strong similarities between MARS ad Opus. (This is to be expected; runtime support for an Opus-like language was the genesis for the MARS project.) Multidisciplinary Opus applications begin with a “leader” that sets up the SDAs, similar to the way MARS ensembles are setup. Opus and MARS both use similar *spawn* operations to invoke working modules with specific resources. Opus provides access to SDA objects without working modules having to explicitly locate the SDA objects on the network. Similarly, the leader module in a MARS ensemble defines I/O paths for working modules at runtime.

Clear differences between MARS and Opus make their target domains quite different. The primary difference is that Opus is a language specification while MARS is a runtime system. With a few extensions to MARS, it would be an appropriate target for Opus to compile to (Opus relies on externally provided runtime support). Additionally, Opus support only extends to SDA-aware codes. There is no legacy code support within Opus.

Opus is a coordination language, not a distributed programming language like Orca or Fx, which makes it more closely related to MARS [Chapman94]. The MARS runtime system is predicated on primitives exclusively designed for *coordinating* modules, rather than *programming* modules.

CHAIMS

The CHAIMS system is similar to MARS, but operates at a significantly higher level of abstraction than MARS. CHAIMS, unlike many of the other coordination systems examined, is close to being a pure composition language, functioning at a higher level of abstraction than task-parallel programming languages and extensions [Tornabene98b]. We outline the philosophy of the CHAIMS system, the differences between the levels of operation of CHAIMS and MARS, and finally examine the fundamental similarities between the two.

CHAIMS programming

The original CHAIMS specification was for a composition language that would be used to build *megaprograms*. *Megaprogramming* is a form of *programming in the large*, where “large” can be viewed on several independent levels. There can be largeness in time (persistence), large variability (diversity), largeness in size (complexity of the program and/or its range of applicability), and largeness in capital investment (infrastructure) [Wiederhold92]. Megaprograms compose *megamodules* into collections capable of greater functionality than an individual megamodule can achieve. A megamodule, as originally specified, was a large, persistent, self-contained object with a consistent interface for querying. Megaprograms are built by composing various megamodules into a whole to achieve a desired result.

CHAIMS is predicated on breaking up the traditional call statement into three main parts: *supply*, *invoke*, and *extract* [Wiederhold92, Tornabene98a]. *Supply* corresponds to traditional argument passing, but does not force an immediate call invocation. It merely provides the input arguments required by a traditional call. The *invoke* statement causes the asynchronous execution of the megamodule with the parameters supplied by a corresponding *supply*. Results are returned to the calling megaprogram by means of *extract*. Various other directives to megamodules allow their status to be examined, inspection of their interface and contents, and execution resource constraints to be placed on megamodule invocations. Information passed between megamodules must be *transduced*: collected, transformed, and forwarded among megamodules and I/O modules [Perrochon97]. In CHAIMS, transduction is currently up to the megaprogrammer.

Differences between CHAIMS and MARS

There are significant differences between CHAIMS and MARS, but they are primarily on the level of abstraction, rather than in philosophy. CHAIMS assumes that megamodules have their own ontology and maintenance, and are simply available to the public for access. In this sense, megamodules are much like object resources in CORBA. MARS is perhaps better characterized as "programming in the small" in this regard. MARS ensembles are formed by collecting stand-alone modules that are not already servers and turning them into servers capable of being composed into ensembles (megaprograms).

CHAIMS assumes that composers (megaprogrammers) generally do not have control over the megamodules they use [Wiederhold92]. MARS assumes that, in general, its modules may be invoked on any participating execution node, and is thus steered toward a different set of problems. At the same time, however, MARS can certainly communicate with persistent modules; MARS simply extends the CHAIMS specification to include dynamic module invocation primitives. This *is* a clear philosophical difference: CHAIMS is not "polluted" with non-compositional features, while MARS is not "limited" to dealing with persistent modules.

MARS is a *runtime system* that provides specific data transport mechanisms to multidisciplinary programs and a simple specification language for composition of those programs. CHAIMS is predicated on communication with existing modules through various methods, including COM, CORBA, Java-RMI, etc. [Tornabene98b]. CHAIMS does not provide data transport mechanisms to previously naïve programs; rather it has a rich set of messaging protocols intended for communicating with existing network-aware servers (megamodules).

The combination of these implementation differences distinguish the MARS problem set from CHAIMS'. MARS is intended for use primarily with local modules. These local modules are controlled by ensemble creators. They can be altered locally and have multiple invocations on arbitrary execution nodes. CHAIMS assumes that megamodules are available *somewhere*, and that megaprogrammers have no control over megamodules and cannot alter them.

Similarities between CHAIMS and MARS

The main difference between CHAIMS and MARS is that they operate at different levels of abstraction. Despite those differences, there are several fundamental similarities between the current specifications of the two systems.

The MARS system uses a wrapper program to make modules operate as mini-servers, existing as long as the user dictates. With the wrapper, previously unsuitable codes can be brought into a multidisciplinary application. The ability to utilize legacy codes, those codes not originally developed with MARS, is important to code reuse. The current CHAIMS specification has a similar notion of a wrapper. The CHAIMS wrapper wraps non-compliant megamodules so they can be used with CHAIMS. The CHAIMS wrapper adds an interface to megamodules that cannot communicate with a megaprogram or that do not correspond to the CHAIMS calling specifications. The wrapper allows *legacy megamodules*, those that are not already CHAIMS compliant, to be used by megaprogrammers. For example, if a megamodule did not support simultaneous access or the split *supply*, *invoke*, *extract* method of calling, a wrapper would make the megamodule appear as if it does so to megaprogrammers [Perrochon97].

The CHAIMS megaprogram is very similar to MARS' leader module. The leader module encompasses the MARS specification language and is purely compositional. The body of the leader code, while expressed in a traditional language (C/C++), is very similar to CHAIMS megaprograms. Both focus on the composition of modules and hide implementation details. Current MARS leaders can direct data from one module to another in an ensemble, but original leader codes had to collect data and retransmit it to working modules. The current CHAIMS system marshals data and retransmits it to megamodules, but the CHAIMS project is working toward direct data flows between megamodules [Tornabene98b].

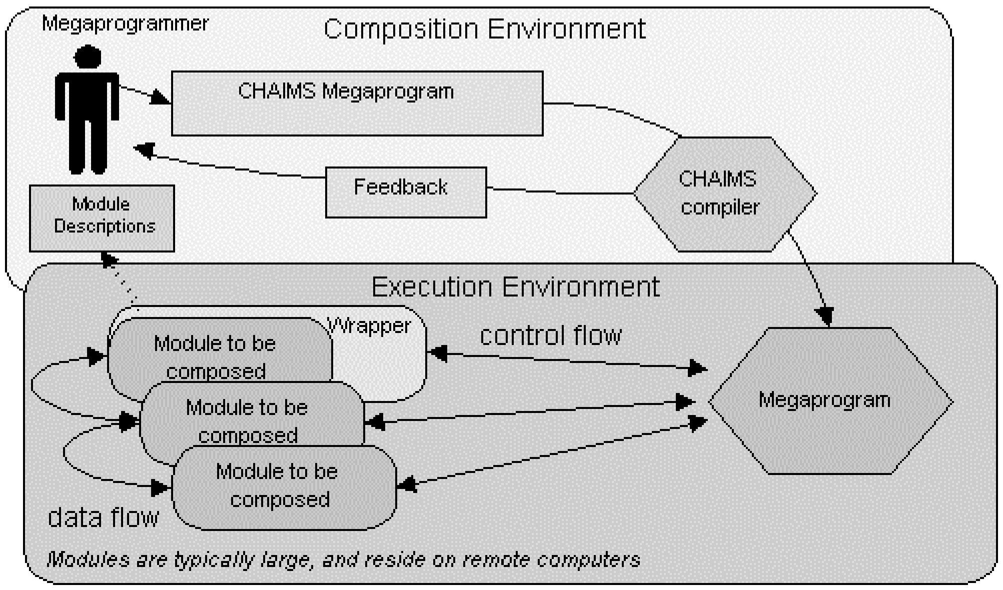

Figure 2. Visualization of CHAIMS composition [Perrochon97]

CHAIMS megaprograms certainly have the ability to be multidisciplinary. An arbitrary set of megamodules from many disciplines can be tied together to solve a particular problem. The thrust of MARS is to support multidisciplinary optimizations. Neither system deals with the *functioning* of component models (megaprograms or working modules), but with the *flows* between and the *coordination* of various models. The assumption made by MARS and CHAIMS is that working modules are oracular; neither affects or alters module-local computations.

As seen in recent project descriptions, visualizations of CHAIMS compositions closely resemble visualizations of MARS ensembles (Figure 2). We outline the primary MARS/CHAIMS parallels in Table 1:

Table 1. MARS/CHAIMS similarities

MARS	**CHAIMS**
Leader **coordinates** working modules	Megaprogram **coordinates** megamodules
Legacy codes are **wrapped** to participate	Legacy codes are **wrapped** to participate
Ensembles can be **multidisciplinary**	Megaprograms can be **multidisciplinary**
Leader codes are **compositional**	Megaprograms are **compositional**
Working modules are considered **oracular**	Megaprograms are considered **oracular**

We have found the closest analogue to MARS to be the CHAIMS system. Even though they operate at different levels, they are very similar in approach and philosophy.

III. The MARS System

The MARS system consists of six main interacting components, the wrapper code, the native communications layer, the Remote Service Layer (RSL), the Remote Service Spawn (RSS), the message processing layer, and the MdoModule layer. There are four main divisions of these components: preprocessing components, runtime library code, support for remote invocation, and high-level objects needed to construct the leader code. The components combine to form a complete framework capable of supporting the multidisciplinary problems outlined in Chapters I and VII. In general, the MARS system works as follows:

1. Component modules necessary to solve a problem are assembled.
2. The modules are wrapped, turning them into specialized MARS servers.
3. A leader is written[5].
4. The leader is executed, causing execution of the component modules in accordance with the specification.

When assembled, the leader and the working modules form a multidisciplinary application. This chapter reviews the components of the MARS system.

The wrapper is the first part of the MARS system. Its job is to turn legacy modules into special MARS servers, capable of communicating with other MARS working modules and leaders. The wrapper installs communication mechanisms in legacy codes and also substitutes MARS I/O routines for those in the original code. Once this is done, a MARS leader code (which is really just a specification for a set of modules) can control the working module by invoking it with the remote invocation tools. Once a module is running, it communicates with its leader through several layers designed to ensure consistent communication between modules. These components relate in the manner shown in Figure 3.

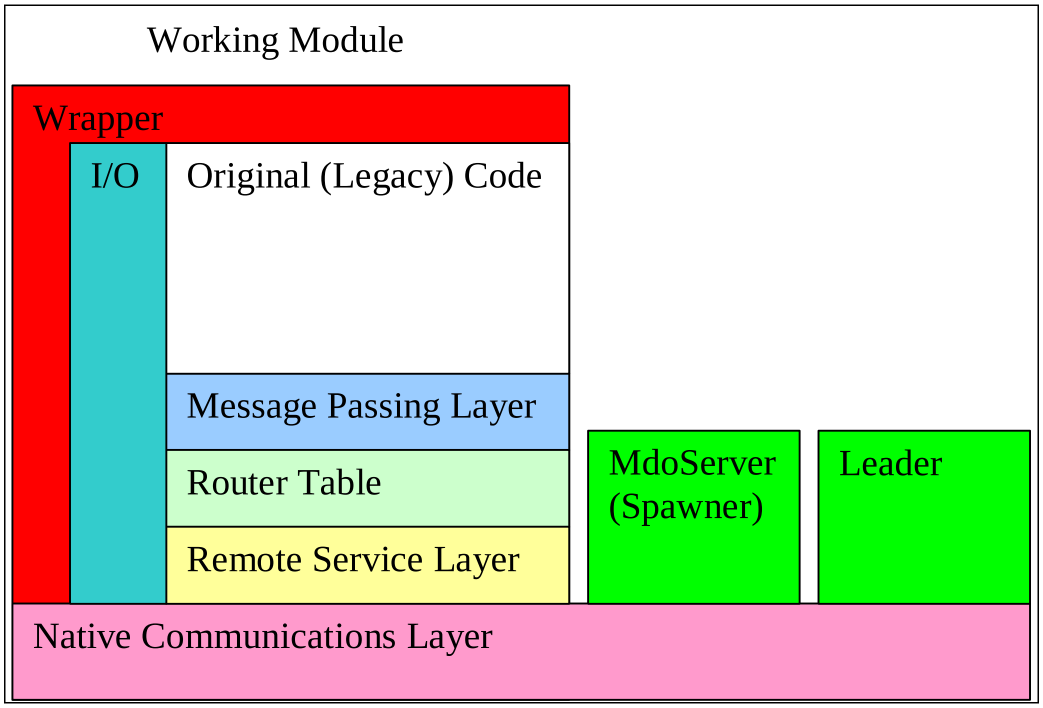

Figure 3. MARS component interactions

[5] The leader is effectively a “map” of the execution and data path to be taken by the assembled modules.

Preprocessing Components

Wrapper

When an ensemble programmer is presented with existing codes to be integrated into a new multidisciplinary application, the first step is to make the codes compatible with the MARS system by transforming them into MARS servers. This is achieved by *wrapping* the existing executables with the runtime routines provided by MARS.

The wrapper begins with the source code for a module and adds essential routines to make the module interact with other MARS modules and leaders. First, the wrapper renames the existing source code's "main" routine (original entry point) to "wrapped__main", or another unique identifier if a "wrapped__main" already exists. The wrapper inserts a new main routine into the program that usurps the original code's entry point and substitutes an initialization code, exit (clean-up) code, and a simple message processing loop to handle incoming messages.

When a wrapped module is invoked, its initialization routines cause the module to immediately *check-in* with the MARS module (leader or otherwise) that invoked it. During the check-in phase, the module tells its invoker its vital identification information, such as its execution host's id, its unique communication id (port number, assuming TCP/IP communications), its nickname (if one is given), and any other identifiers embedded at wrap time. This check-in communication is done via the Remote Service Layer (RSL).

After check-in, a wrapped module simply waits for incoming connections and/or messages. When a message is received, it is processed by the runtime system which is linked in after the wrapped source code is compiled.

The wrapper has functionality built in to allow users more control over wrapped modules than is available in the original (unwrapped) module. For instance, with specific command line options passed to the wrapper, programmers can add module entry points for subroutines that were previously externally inaccessible. For instance, if the programmer knows that operations are safe, s/he could invoke those subroutines within a module and receive output from those subroutines. For example, a stand-alone module would have previously restricted the module's execution flow to cover only the code invoked by the main routine. A wrapped code can invoke *any* arbitrary subroutine from the original source code, at any desired point. This ability might violate the black box principle of the original code, and users of this functionality must be aware of the potential side-effects caused by arbitrary routine invocations.

Wrapped programs can also run in *stand alone* mode. Stand alone mode is simply the invocation of a wrapped module as it would have been previously invoked (before wrapping), without the Remote Service Spawn (RSS) layer. For example, a wrapped program that was previously invoked by typing "./foo –o bar" on the command line would still give the same output in the same form as the original non-wrapped code. Stand alone mode facilitates component testing, and allows a single copy of a module to perform within the MARS framework or without. This "single source" capability is extremely important in simplifying code maintenance. Programmers can tweak a single source that runs in stand-alone *and* MARS modes, without having to post changes to multiple sources.

Control Loop

Virtually all communication primitives in the MARS system are asynchronous. Users can easily construct synchronous calls by blocking on return values from calls that have returns (e.g., `RSL->TERMINATE_AND_REPLY()`). The control loop added to a wrapped module consists of a simple message receive/process loop, augmented with the ability to accept new incoming connections. The typical control loop is similar to the following (in pseudo-code):

```
initialization_code();
status = RUN;
while (status != EXIT) {
  RSMessage = poll_for_message();
  status = process_message(RSMessage);
}
exit_code();
```

The initialization code takes care of the check-in procedure. To some extent, the check-in procedure will be protocol dependent (e.g., an Intel Paragon's native messages or Linda's tuples would require different informational fields from a Linux machine using TCP/IP). Currently, only TCP/IP support is implemented, though we have used TCP over several different mediums, including 10/100baseT Ethernet and fiber-optic ATM, and it is available on most systems, including those without native TCP/IP support.

The control loop runs until the module is explicitly directed to exit, at which point its exit status is returned to the module that ordered its termination. Note that module termination is not necessarily called exclusively by the leader that invoked the module.

In many cases, especially with complex conditional execution paths, it is desirable to allow arbitrary modules to force their peers to exit.

File (I/O) Layer

The critical issue with the wrapper is I/O. Many potential candidates to become modules in an MDO use arbitrary output schemes that use multiple files, in multiple (often proprietary) formats that are without obvious meaning to an outside observer. Such programs can be extremely difficult for a developer to understand and integrate into a multidisciplinary optimization.

The wrapper must be able to convert simple I/O calls automatically to some other form that can be utilized in a collection of different modules[6]. Disk-based I/O routines are by far the most common, especially in scientific codes. Simulation data is generally pushed to disk and later analyzed or used by other programs.

The problems with integrating disk-based I/O routines into a MARS module cluster are many. Frequently, execution nodes will not share the same file system, making it extremely difficult to share files conveniently. Also, if wrapped modules have hard-coded absolute paths to input and output files, it can be very difficult to distribute modules to arbitrary execution nodes. Finally, to take advantage of task parallelism, disk-based I/O times should not dominate processing time. If it does, the time to read,

[6] We are currently investigating the SmartFile approach to understanding data [Haines95b]. More information about SmartFiles is provided in Chapter VI.

write, and export files from a given node will dominate the process time, negating one potential benefit of the MARS system.

To combat these problems, we use a special I/O layer written to abstract files from physical devices (disks) and substitute existing network communication channels instead. For example, a pre-wrapped I/O call might write an integer to a file called "out.dat". Each write involves a call to an operating system defined primitive (such as `write() or printf()`) that accesses a mechanical device (disk) and has an associated latency cost that is generally high compared to a memory or network write.

Instead of necessarily committing data to disk, our I/O layer transmits the data *somewhere*. When a module is invoked, its data streams do not yet go anywhere! Some time after the check-in procedure has finished, a module is instructed to send its data to an arbitrary location, be it memory, the network, or to disk. If no explicit information is provided, the module's *original* I/O destination is used. This means that the module, when invoked in stand alone mode, still understands its original data paths, and still functions properly.

The file I/O layer is implemented as a C++ class. There are five basics primitives for binary type I/O:

```
int c_open(char * name, io_type type);
int c_close();
int c_flush();
int c_write(const void * buff, int size);
int c_read(void * buff, int size)
```

The I/O layer will write to disk or to a connected socket, depending on the module's invocation method ("stand alone" or part of an ensemble). The I/O layer's function is similar to an operating system's disk-based I/O, with a "flush" mandating that data be pushed to disk or to the network and allowing data to be committed at arbitrary intervals otherwise.

Runtime System

The runtime system has four main parts: the native communications layer, the Remote Service Layer (RSL), the Router Table, and the Message Processing Layer (MPL). To remain consistent across platforms, the RSL can be built upon whatever native communications layer is available; for this implementation of MARS, we chose TCP/IP as it is the most widely-available transport protocol today.

The Native Communications Layer

The native communications layer is the foundation of all message passing in the MARS system. In this implementation, we rely on TCP/IP sockets for low-level communications, but the messages can be easily ported to other protocols because the low-level protocol simply passes RSL-encoded messages. An RSL-encoded message is simply a standardized data packet with a header block that describes how the data is packed.

The native communications layer must provide simple primitives to the RSL layer, while the RSL layer is responsible for all data encoding and transmission decisions. The assumptions made about the data channel required to support the RSL layer are that the low-level system is *reliable* and that the messages are *ordered* along a given channel.

TCP/IP is both reliable and ordered for a connected socket. Given these requirements, UDP/IP would not be suitable for our RSL because it is unreliable and unordered.

In environments with connectionless protocols, such as the Intel Paragon or the Linda system, simple integer counters local to a given execution node would suffice for message ordering (with identification of the transmission and target modules, of course). A total global ordering of messages is not required; simple ordering of messages between nodes (on a given real or virtual channel) is all that is necessary.

Our system implements the following essential, basic public primitives to access the native communications layer:

```
// connect (as client) to a server
int connect (char *toHost, int toPort);

// check for possible requests to connect (non-blocking)
bool pendingConnect (int secs = 0, int usecs = 0);

// accept (as server) a connection from a client socket
// (blocking)
int accept ();

// close connection(s)
void close (int connectId);      // one connection
void close ();                   // all connections

// check connection status
bool isConnected (int connectId);

// write data over a connection
int write (int connectId, void *buf, int bytes);

// check for possible messages (non-blocking)
bool pendingMessage (int connectId);
bool pendingMessage (int *connectId, int secs = 0,
     int usecs = 0);

// read data from a specific connection
int read (int connectId, void *buf);
void *readAlloc (int connectId, int *size);
```

These primitives are essential to the RSL layer functionality. They provide a means to establish a "connection" (be it a virtual or connected message medium, such as TCP/IP) and to pass messages over such connections. In the TCP/IP implementation, the connections are managed in a *connection table* that is a component of the *socket class*. The connection table manages the virtual circuits formed by connecting two TCP/IP sockets. If MARS were ported to run in another environment, such as Linda, the connection table could simply track IDs associated with tuples that would designate which virtual circuit a message belongs to.

In our system, the low-level communications package includes header information used to form message packets and to control transmission characteristics. In many instances, such as when streaming data leaves one module directly bound for the input stream of another module, such overhead is unnecessary. Such a circuit, used without header information, is referred to as a *raw* circuit. The following advanced public primitives were implemented to leverage the ability of the native communications protocol to quickly transmit data, without the need for extra header information:

```
//return connection mode ("raw" or "packet")
bool pktmode  (int table_id);

//changes connection mode to TS_RAW_MODE
void make_raw (int table_id);

//changes connection mode to TS_PKT_MODE
void make_pkt (int table_id);

//reads from a raw connection into ptr
//will read up to "size" bytes, returns status
int raw_read     (int table_id, char * ptr, int size);

//blocks until size bytes are read, or error occurs,
//returns status
int raw_blk_read (int table_id, char * ptr, int size);

//determines bytes available for reading
```

```
int raw_blk_peek (int table_id, char * ptr, int size);

//writes to raw connection without any header information
int raw_write    (int table_id, char * ptr, int size);
```

These advanced primitives may not be appropriate for some native transmission protocols (other than TCP/IP) and are not necessary for the RSL to function properly. True RSL messages always have header information, indicating message type, message source, etc. These primitives are appropriate where applicable, but not necessary for complete system functionality.

The native communications layer in this case (TCP/IP) had one main deficiency. When a certain amount of data was sent from module “A” to module “B”, if module “B” did not consume the information by reading it from the network, module “A” was forced to block on a send until module “B” read the waiting data. On many systems, the amount of data that could be buffered was unusually low, i.e. 16k bytes. To remedy the situation, we added a *line buffer* between the modules that required it. The buffer is a multi-threaded application that queues incoming data until it can be safely sent to the consuming module.

The buffer program is an implementation of the classic producer/consumer problem. There is a reader thread and a writer thread corresponding to each direction of data flow (so there are four threads, in total). The reader thread constantly places data into a buffer, while the writer thread constantly passes data down the network. In this way, if a writer thread blocks because a particular module is not consuming data, the corresponding reader thread can still place data into the buffer, thus permitting both modules (producer and consumer) to continue without blocking.

There is a tradeoff to this buffering system: messages must be copied an extra time. Alternate approaches include increasing OS buffer sizes and various forms of flow control. Increasing OS buffer sizes simply delays the eventual time until buffer overflow, while increasing OS overhead. Various forms of flow control have yet to be examined, and are an ongoing research issue.

Remote Service Layer (RSL)

The Remote Service Layer (RSL) is the primary method of communication between modules. RSL communication is ordered between modules along virtual connections because the native communications layer is similarly ordered. RSL messages are asynchronously transmitted and received, without explicit return values in the general case. Special fields in an RSL message header facilitate synchronicity (if desired), but this synchronicity comes from how the message is processed in the Message Processing Layer. RSL messages consist of two parts: a well defined header and a corresponding binary data-blob sent immediately after the header. An RSL message header is an object of the following form[7]:

```
typedef struct RSmessage
{
  char from_name[100];  //not required, used by MPL
  int  from_port;       //not required, used by MPL

  char type[100];       //request type
  char subtype[100];    //request sub-type, if necessary

  //packed arguments are word-aligned and can be accessed
```

[7] RSMessage are only *sent* and *received* by the RSL layer (no message processing is performed by the RSL layer).

```
    //directly
    int  num_args;                    //number of packed args
    int  args    [RS_MAXNUMARGS]; //offsets of packed args
    int  arg_size[RS_MAXNUMARGS]; //sizes of packed args
  } RSmess;
```

The RSMessage type has fields `type` and `subtype` that indicate the purpose of the message. RSMessage types are any string recognized by the Message Processing Layer, and are simple null-terminated strings. `num_args` indicates the number of arguments in the RSMessage, while `args[]` and `arg_size[]` indicate how those arguments are to be accessed. The type and meanings of the arguments are determined by the context provided by the `type` and `subtype` fields.

An RSMessage is routed in a manner similar to the native communications messages. A corresponding RSMessage connection table is built on top of the native communications layer's connection table. The RSL adds several features not available in the native connection table, however. For instance, the RSL allows connection *naming*, so that channels may be accessed by their "name" rather than a connection ID. Also, circuits have the ability to name themselves, further advancing the notion of asynchrony. For example, a naïve unnamed comminations protocol would require that clients connect to a server in a specific order, forcing that high level of synchronization upon the entire multidisciplinary optimization. It might look something like the following:

```
int a = accept(“connection from module a”);
int b = accept(“connection from module b”);
int c = accept(“connection from module c”);
run(a);  run(b);  run(c);
collect_data(a, dataA);
collect_data(b, dataB);
collect_data(c, dataC);
```

In such an example, nothing could begin until modules a, b, and c connected to the server, and the connections would necessarily have to occur in that order. With naming installed, the following code could be used, without the synchronization forced by the mandatory connection ordering:

```
accept(“any”);
accept(“any”);
accept(“any”);
run(“a”);  run(“b”);  run(“c”);
collect_data(“a”, dataA);
collect_data(“b”, dataB);
collect_data(“c”, dataC);
```

At first glance, the syntactical differences between the two examples do not fully illustrate the semantic differences. Notice, though, that the first example requires that “module a” connects first and is thus assigned the unique identifier “a”. Furthermore, all later operations depend on the identifier “a” pointing to a *specific* structure. With the naming scheme, a program can receive an arbitrary set of incoming connections and process them in an arbitrary order, depending only on the runtime instructions of the program. This dynamism facilitated by connection naming will be explained further in Chapter IV. Applications/Results, where will look at how dynamic scheduling can be accomplished by using names instead of static connection identifiers.

RSL messages are sent using the `rsr` method which can be routed via connection id or by name:

```
//rsr by connection id, returns status
int rsr (int destination, char * method, int nArgs, ...)

//rsr by connection name, returns status
int rsr (char * dest_name, char * method, int nArgs, ...)
```

The variable argument list (denoted "`...`") is a very important component of an RSL message. We pack the data as efficiently as possible, but align each argument of the argument list, regardless of its size, on a word boundary (the base of the alignment corresponds to the RSMessage header information). We discovered that several important architectures, including the MIPS, would not properly handle certain native data types (`int`, `double`, etc.) that are a word size or larger, on the receivers' end, unless they were pre-aligned on word boundaries. For example, the following code fragment would fail[8] on the second cast on many machines:

```
char buff[20];
int i = *((int*)(&(buff[0])));   //succeeds
int j = *((int*)(&(buff[1])));   //fails
```

To combat the problem of receiving arbitrary data types in an RSL message, each argument is independently aligned to the largest word size among participating machines (at a maximum cost of `sizeof(word)-1` bytes per argument).

Router Table

The Router Table is an advanced data structure for installing arbitrary names in a module's I/O routines. The Router Table provides a bridge between the Remote Service Layer and File I/O Layer. An unaltered, non-wrapped module may expect to receive incoming data from a file, say "`foo`". Instead of demanding that the wrapped module constantly write to a fixed channel "`foo`" (which could go to disk or network), the Router Table allows the destination "`foo`" to be changed dynamically, unbeknownst to the

[8] The code will compile, but fails at *runtime*.

module using “`foo`”. This ability is extremely powerful. Unlike the languages using compiled communication paths, like Fx, MARS modules never need to know where their inputs come from until after they are invoked. Likewise, their outputs can be similarly assigned at runtime and can be changed multiple times during their execution lifetime.

For example, during its first run, a module might write its output to “`foo`”, and “`foo`” could go to a disk file named “`out.dat`”. During its second run, the program again writes its output to “`foo`”, but “`foo`” now points to the input stream of another module elsewhere on the network *and* a disk file called “`run2.dat`”. During a third run, “`foo`” may be superfluous, and the data written to it may be discarded. All of these changes occur without altering the running module, except for a simple message that replaces the location in the Router Table pointed to by “`foo`”. The power of this dynamism is shown further in Chapter V. The Router Table is managed exclusively by the Message Processing Layer.

Message Processing Layer (MPL)

The Message Processing Layer is the most critical component of the MARS system. It is the layer that translates and implements all requests in all RSL messages. It modifies connection tables, invokes native procedures within wrapped modules, and establishes data connections. There is only one public method (other than the constructor and destructor) that the MPL understands:

```
int process(rsl * transport, char * message);
```

The "`rsl * transport`" is a pointer to the actual RSL object and is only essential if a module has multiple transport systems. We do not foresee any legacy applications (those that would wrapped) requiring multiple network transports, but understand that it may be useful to original application designers. The "`char * message`" is actually a pointer to an `RSMessage` type, but we have found that some compilers prefer an external cast to `char` and a re-cast inside the method to the `RSMessage` type. Also, and more importantly, we can receive a `RSMessage` type into a `(char *)` buffer and process it without the module requiring knowledge of the `RSMessage` type at compile time. As such, it is very easy to alter the `RSMessage` specification without recompiling the source for the main module.

The `process()` routine understands many requests, and has private methods for each type. There are currently fifteen known message types and a handler for unknown messages. The unknown message handler simply relays the original message back to the sender, with an error status indicated. The messages processed by the MPL compose the specification language.

The messages that can be processed instruct a module to perform the various actions required to run smoothly. If module "A" determines that module "B" should establish a new output connection to module "C", module "A" sends a message to "B" indicating that request. The Message Processing Layer understands that to fulfill the request, it should call the appropriate routines from "B's" RSL and force the connection to module "C". When it is time for a module to execute its `wrapped__main` routine, the MPL understands where the entry point is and invokes the routine.

Most mundane functions are also performed by the Message Processing Layer. Log files are opened and written to by the MPL. The level of logging to be performed by the lower layers (such as File I/O Layer logging) are altered with calls to the MPL. It is the layer that understands how to control all the other layers and the only level at which modules can communicate directly.

Logging

Logs are important part of any large scale programming endeavor. In order to troubleshoot and tune runtime systems, especially with dynamically changing schedules and data paths, it is important to understand the state of particular modules and how the effects of their states affect the global state of a collection of modules.

We have implemented logging primarily in the MPL and the File I/O Layer. The logging can be done at several *levels*, depending on the log event granularity desired by the programmer. Log levels can be changed at runtime, rather than just at compile time. This flexibility allows the user to turn off all logging in stable codes, ensuring optimal performance, and (at a different time) the user could turn on high log levels to troubleshoot the same codes if problems arise.

Remote Invocation System

The remote invocation system for MARS was built from scratch. We chose to use a simple TCP/IP based method, rather than require users of MARS to have access to an emerging standard, like commercial implementations of CORBA. (Also, at the time this project was started, different CORBA vendors' implementations were often not compatible.)

Remote Service Spawn (RSS)

To invoke a module on a remote execution node, a call is made to the Remote Service Spawn. The RSS call for TCP/IP looks like this:

```
int spawn(char * checkin_server, int checkin_port,
          char * spawn_server, char * spawn_prog,
          char * args = “”);
```

This call can be abstracted to the form:

```
spawn(hostType * execution_host_id,
      hostType * spawn_host_id, char * module_to_spawn,
      char * arguments);
```

This form can be used with invocation methods other than TCP/IP. At a minimum, a call to `spawn()` requires the name of a module to execute, the node to execute the module on, and the arguments to the module (if any). After careful consideration, the `spawn()` command has been kept separate from the RSL and native communications layer to allow heterogeneous *invocation* methods to exist without altering the *communication* methods. Invocation and communication are radically different tasks.

The spawn command invokes the module on the specified node and passes to that module the extra information required for the check-in procedure that is embedded in a wrapped code. The RSS always logs invocations and the return status from a `spawn()` (invocation) call.

MdoServer

The MdoServer is the code that actually performs the invocation of a module; it handles the spawn call from the RSS. In this implementation of MARS, the MdoServer is invoked by the UNIX `inetd` process when a call is made to the TCP/mux port (port 1), and “mdo_server” is the service requested. (The RSS knows to connect to the TCP/mux port and to request the “mdo_server” service.)

The MdoServer accepts the arguments provided by the RSS, and forms a command string with the module to be invoked, the check-in arguments, and any further arguments to the module. The command string is passed to an `exec()` call, and the MdoServer’s executable image is replaced with the requested module’s.

The MdoServer is a simple C code that only depends on the `exec()` call. It is a simple program that could be replaced with another invocation method, such as one dependent upon CORBA.

TCP/mux code

The TCP/mux service is available on many, but not all, UNIX operating systems. For instance, SGI’s Irix 6.3 operating system has TCP/mux as an internal service, but the same service is lacking in Sun’s Solaris 2.5.1 (both OSes were used during testing). As such, we include a simple TCP/mux handler with the MARS distribution. For each execution node requiring it, it can be compiled locally and installed by adding a single line to the `/etc/inetd.conf` file. Documentation for doing so is included in the MARS distribution.

The code we include is based strictly on RFC 1078 [Lottor88]. It does not affect system security or provide any advanced features, but it does provide a mechanism by which modules can be easily spawned. Also, using the TCP/mux option for spawning modules means that there does not need to be an MdoServer running constantly, and no "well-known" TCP/IP port is required.

This generic approach is very different from having to "register" all participating computers and running a special daemon which handles spawns (as is done in PVM). Like PVM, in CHAIMS, all available megamodules are registered with a central clearinghouse. We avoid this problem by having the leader specify which modules it wants, and making the leader responsible for invoking existing modules. With the types of applications built for MARS, this makes sense. Modules can be freely relocated without having to alter or update the MdoServers, which is not true for other common spawning mechanisms.

Leader Codes

The *leader* code is the module that coordinates all other modules in a multidisciplinary program built by combining the functionality of modules from different disciplines. In a program built from multiple *modules*, the leader is analogous to a creature's brain, while the other modules perform the other required bodily functions, at the instruction of the brain. Leader codes can range from the simple to complex, and consist of two main parts: the *setup stage*, and the *execution* stage. There could be other stage designations, such as *termination*, but the setup and execution are of primary interest. The specification language portion of MARS consists of leader and sub-module connection setup and execution.

The setup stage is where the modules that compose the “body” of an ensemble are designated. They are *added* to a MdoModule object, the primary object manipulated by the leader. Modules are added by specifying their location on the network and any necessary I/O paths that they are initialized with. The following fragment illustrates both the actual creation of an MdoModule called “leader” and the addition of two working modules to the leader:

```
//setup leader module
MdoModule * leader = new MdoModule(transport, “Leader”);

//setup “smooth” sub_module
int smooth = leader->add_module(“smooth”);
leader->set_host(smooth, “ghengis.cs.uwyo.edu”);
leader->set_path(smooth,
  “/export/home/mdo/matrix-loop/smooth”);
leader->set_args(smooth, “”, 0);
leader->set_file(smooth,(char*)”data.set”, OUT);

//setup “test_for_termination” sub_module
int test = leader->add_module(“test”);
leader->set_host(test, “conan.cs.uwyo.edu”);
leader->set_path(test,
  “/export/home/mdo/matrix-loop/test_for_termination”);
leader->set_args(test, “”, 0);
leader->set_file(test,(char*)”data.set”, IN);
leader->set_file(test,(char*)”data.set”, OUT);
```

This example sets up a leader module, aptly named “leader”, and adds two sub-modules, “smooth” and “test_for_termination”, to the set of modules controlled by “leader”. The first call to `leader->add_module()` creates a new MdoModule object pointed to by `leader`. This object will be used to reference a working module. After that, calls to `set_host()`, `set_path()`, `set_args()`, and `set_file()` determine on which *execution node*[9] the module should be executed, where the executable image for the module is located on that execution node, what arguments the module should receive, and what I/O paths to use by default. Notice that the two sub-modules are added to “leader”.

To connect the module’s I/O streams and invoke the modules, the following code is used:

```
leader->link(“data.set”, “data.set”);
leader->setup(spawn);
```

The link call may seem confusing with the name `data.set` appearing twice, but the `leader->link()` call is smart enough to understand that the same name can be used in multiple instances and matches the `(“data.set”, IN)` to a corresponding `(“data.set”, OUT)` in another sub-module automatically. Programmers can explicitly designate connections, and it is recommended they do so, but this example shows the *minimally* requisite information. The `leader->setup()` call makes calls to an RSS routine appropriate for spawning the modules and creates appropriate RSL messages for naming I/O connections. The `leader->setup()` call also requests that sub-modules form connections among themselves[10].

[9] An execution node consists of the resources required to run a program, generally thought of as a CPU or machine.

[10] The programmer is not responsible for recalling the underlying RSL routines. They are invoked automatically by the SubModule object.

Because of the design of the SubModule object, only this minimal set of information was required to set up an actual leader with two sub-modules and all the necessary connections. There are many connections hidden from the leader programmer, including RSL connections from the leader to each sub-module for sending further control messages to the sub-modules. The programmer does not have to worry about these, either, as they are created and named automatically.

The following sample shows how easy it is to make something useful happen (assuming of course the sub-modules do something *useful*):

```
//make smooth start procedure called “read_in_data”
leader->start(smooth, “read_in_data”);
while (done == 0) {

  //make smooth run procedure originally called “main”
  leader->start(smooth);

  //make test run procedure originally called “main”
  leader->start(test);

  //get result of test_for_termination here
  c_fscanf(file, “%d”, &done);
}

//make smooth run procedure called “write_out_data”
leader->start(smooth, “write_out_data”);
```

The first line of code causes the module called “smooth” to execute an internal procedure called `read_in_data`[11]. After that, the `wrapped__main` routine of smooth is invoked, by calling to `leader->start(transport, smooth)`, without an addition field. Afterwards, the sub-module “test” has its `wrapped__main` invoked. A

[11] Remember that the wrapper has the ability to wrap *any* original internal procedure to allow execution at any time. The original entry point (“main”) is not the only routines that can be invoked.

status variable, `done`, is refreshed to determine if the loop should continue. Notice the `c_fscanf()` used to read in the variable `d`. The variable can come from a disk-based file, or from another source, such as `(“data.set”, OUT)`.

When the loop is complete, there is a final call to `leader->start (transport, smooth, “write_out_data”)`, another method inside the module “smooth”. Cleanup and termination of the sub-modules is very simple:

```
leader->end();   //that’s it
```

Viola! All loose ends are cleaned up with this one call. The leader understands where its sub-modules reside and generates the calls necessary to effect graceful exits. Wrapped modules also track which connections they initiate and close them as well.

This sample leader code is a demonstration of all that is required to invoke and control two wrapped sub-modules. Leaders can be less complex or much more complex. With the proper use of condition variables, and with steering code added to leaders, any amount of execution control and dynamism is possible.

Summary

The primary goal of the MARS project is to build a runtime system to support *multidisciplinary applications*. These applications are formed from collections of modules that often include *legacy* codes, not just newly created modules. While MARS supports legacy codes, it is still an excellent *authoring system* for distributed codes. The working modules that form an ensemble cooperate in a *heterogeneous* execution environment, frequently composed of clusters of workstations.

MARS modules can be invoked as part of an ensemble or alone. As such, MARS supports *supports single-source* coding, making code maintenance easier for ensemble creators.

The runtime system supports *dynamic invocation*, *termination*, and *realignment* of modules. Module distribution is a simple task; it does not require a registration step like CORBA and other invocation strategies. While dynamic module realignment is a simple task, dynamic *redirection of the user data* passed between modules is possible as well. Primitives for *conditional execution* also exist in MARS.

Even with all these control considerations, *performance* is still central to MARS' implementation. MARS does not add significant overhead for its control or data communications to multidisciplinary applications. MARS can be used to distribute and parallelize collections of otherwise stand-alone modules into cooperative ensembles.

MARS is designed to take expert knowledge embodied in multiple computer programs and glue those programs together. The program ensembles formed by this process are used to solve problems larger than any component module can solve.

IV. Applications/Results

The chapter demonstrates four test ensembles constructed with MARS. The first code, similar to the sample leader code in Chapter III, demonstrates a simple module that does matrix smoothing coupled with a second module that tests the matrix for a predefined level of smoothness. The code introduces the reader to the basic MARS process and presents performance benefits gained by moving I/O from disk to direct message passing between cooperating modules.

The second code presents MARS used to implement a pipeline parallelism example. Three graphics filters are applied, in succession, to a series of images. The three filters, which increase sharpness (*is*) in an image, posterize (*po*) an image, and reduce noise (*rn*) in an image, are wrapped to become modules and then fed multiple files from a leader code. The example shows the ease of constructing what could otherwise be a complex pipeline and how that pipeline can dramatically improve performance.

The third example uses the identical wrapped modules from the second example in an advanced arrangement to greatly enhance the performance of the graphics filters. This extended pipeline parallelism test demonstrates how MARS can be extended to include dynamically module invocation to overcome performance bottlenecks. This example shows how ensembles can be easily rearranged without re-coding them.

The fourth and final example uses an actual scientific research code taken from the biological sciences. In this example, we outline some of the limitations of the current MARS version, but also demonstrate dramatic performance increases using our system. This test ensemble, used in actual research, is a litmus test for the applicability and performance of MARS on legacy modules. This code is multidisciplinary, and all of the modules were designed before the conception of MARS.

Simple Matrix Example

The first sample program constructed using MARS shows the basic steps required to get an ensemble up and running. Also, we show how the MARS system can use one module's outputs to form a test condition that determines whether we execute another module.

The basic program that will be wrapped here is a routine that smoothes an **N**x**N** matrix. The initial program (before wrapping) looked like this:

```
//data structure to store matrix
int * data1[MATRIX_SIZE];

//routine to smooth matrix
int smooth(int x, int y, int i, int j, int * * data1,
           int *total, int *count)
{
  if (((x+i >= 0) && (x+i < MATRIX_SIZE)) &&
      ((y+j >= 0) && (y+j < MATRIX_SIZE))) {
    *total += (data1[x+i][y+j]);
    *count += 1;
    return 1;
  }
  return 0;
}

//routine to read in matrix
int read_in_data (void) {
  int x ,y;
  FILE * file;

  for (x = 0; x < MATRIX_SIZE; x++)
    data1[x] = new int[MATRIX_SIZE];

  file = fopen("/export/home/mdo/loop/MDO/data.set", "r");
  for ( x = 0; x < MATRIX_SIZE; x++)
    for ( y = 0; y < MATRIX_SIZE; y++) {
      fscanf(file, "%d ",&(data1[x][y]));
    }
  fclose (file);
  return 1;
}

//routine to write out matrix
int write_out_data (void) {
  int x, y;
  FILE * file;

  file =
   fopen("/export/home/mdo/loop/MDO/result.data.set", "w");
  for ( x = 0; x < MATRIX_SIZE; x++)
    for ( y = 0; y < MATRIX_SIZE; y++) {
      fprintf(file, "%d ",data1[x][y]);
    }
  fclose (file);
  return 1;
}

//original code entry point
int main (void)
{
  File * file;
  int x, y;

  int * data2[MATRIX_SIZE];
  for ( x = 0; x < MATRIX_SIZE; x++)
    data2[x] = new int[MATRIX_SIZE];
  read_in_data();
  for ( x = 0; x < MATRIX_SIZE; x++)
    for ( y = 0; y < MATRIX_SIZE; y++) {

      int total = 0;
      int count = 0;

      for (int i = -1; i <= 1; i++)
        for (int j = -1; j <= 1; j++)
          smooth(x, y, i, j, &data1[0], &total, &count);

      data2[x][y] = total/count;
    }

  for ( x = 0; x < MATRIX_SIZE; x++)
    for ( y = 0; y < MATRIX_SIZE; y++)
      data1[x][y] = data2[x][y];
  file = open("data.set", "w");
  for ( x = 0; x < MATRIX_SIZE; x++)
    for ( y = 0; y < MATRIX_SIZE; y++) {
      fprintf(file, "%d ",data2[x][y]);
    }
  close (file);
  return 1;
```

```
}
```

The code is very simple. A matrix is read from disk, smoothed, and then written out to disk. The routine `read_in_data()` is simple enough that it could have been written inline, but we did not do so to make a further point about local routines. Likewise, `write_out_data()` is not used in the original program, but it will be used to demonstrate how wrapped routines can be called arbitrarily and used to make MARS leader programs simpler.

For now, it is not important to follow every detail of the matrix smoothing program. Simply understand that there are four routines, `smooth()`, `read_in_data()`, `write_out_data()`, and `main()`. The first step to converting the original code to a MARS module is the wrapping process. We will wrap the original `main()`[12] and the `write_out_data()` routines with the following command:

```
wrap smooth.cc write_out_data –o smooth.wrap.cc
```

The wrapper is currently rather unsophisticated. The first argument to `wrap` must be the name of the source file to be wrapped. Subsequent arguments are the names of internal functions that require external references (i.e., `write_out_data` will have a new entry point). Finally, if a "`-o`" is given, the argument following it is the name of the output file (otherwise a default naming scheme is applied). After wrapping, code is added to the original program, primarily consisting of calls to the initialization routines

[12] Which is always done by default and automatically.

and additional entry points specific to the module (those that could not be pre-compiled into the runtime library).

After wrapping, the original code has changed very little. Critical sections of the wrapped code would appear similar to the following:

```
//message transportation mechanism
rsl * trans;

//this was the original main, almost identical
int wrapped_main (void)
{
  File * file;
  int x, y;

  int * data2[MATRIX_SIZE];
  for ( x = 0; x < MATRIX_SIZE; x++)
    data2[x] = new int[MATRIX_SIZE];

  for ( x = 0; x < MATRIX_SIZE; x++)
    for ( y = 0; y < MATRIX_SIZE; y++) {

      int total = 0;
      int count = 0;

      for (int i = -1; i <= 1; i++)
        for (int j = -1; j <= 1; j++)
          smooth(x, y, i, j, &data1[0], &total, &count);

      data2[x][y] = total/count;
    }

  for ( x = 0; x < MATRIX_SIZE; x++)
    for ( y = 0; y < MATRIX_SIZE; y++)
      data1[x][y] = data2[x][y];

  //notice the c_open() rather than open()
  file = c_open("data.set", "w");
  for ( x = 0; x < MATRIX_SIZE; x++)
    for ( y = 0; y < MATRIX_SIZE; y++) {
      //notice the c_fprintf() rather than printf()
      c_fprintf(file, "%d ",data2[x][y]);
    }
  //notice the c_close() rather than close()
  c_close (file);

  return 1;
}
```

```
/*****************************************************
BEGIN ADDED BY WRAPPER
*****************************************************/
#include “message.h”
#include “rsl.h”
#include “rt.h”

//Process invalid requests
int invalid_request(rsl * sock, char * buffer)
{
  int table_id;
  table_id = sock->find_id_by_address(((RSmessage *)
     buffer)->from_name, RSmessage *) buffer)->from_port);

  sock->rsr(table_id, “invalid_request”, 1, buffer,
     sizeof(buffer));

  return 1;
}

//INVOKE OTHER (USER-DEFINED) ROUTINE
char * invoke_other(rsl * sock, char * buffer, int* size)
{
  RSmessage * tempm = (RSmessage *)buffer;
  char * info = NULL;

  *size = 0;

  //DYNAMIC CODE
  if (strcmp((char*)&buffer[tempm->args[0]], “main”) == 0){
    wrapped_main();
  }
  else if (strcmp((char*)&buffer[tempm->args[0]],
     “read_in_data”) == 0) {
    read_in_data();
  }
  else if (strcmp((char*)&buffer[tempm->args[0]],
     “write_out_data”) == 0) {
    write_out_data();
  }
  //END DYNAMIC CODE
  else {
    invalid_request(sock, buffer);
  }

  return info;
}

/*****************************************************
 *    “NEW” MAIN ROUTINE
 *****************************************************/
```

```
int
main(int argc, char ** argv)
{
  int i, j;
  RT * rt;
  msg_cli * msg;
  char buffer[20000];
  char server[50];

  int port;

  trans = new rsl;
  rt    = new RT();
  msg   = new msg_cli(rt, rt->get_id());

  change_mode();  //running in stand-alone or server mode

  /****************************************************
   *  Initialization and Check-in procedure
   ****************************************************/
  //connect to leader
  i = trans->rsc((char*)argv[1], (int)atoi(argv[2]));
  strcpy(server, trans->host_name());
  port = trans->port_num();

  /****************************************************
   *  Msg loop
   ****************************************************/
  int quit = (_FINISH + 1);
  while ( quit != _FINISH ) {
    j = trans->poll_accept();
    j = trans->poll_message(&i, buffer, 1);
    if (j > 0) {
      quit = msg->process(trans, buffer);
      if (quit == _RUN)
        invoke_other(trans, buffer, &j);
    }
  }

  return (1);
}
```

There are few changes made to the original code by the wrapper. For the most part, the wrapper *adds* code. The first addition made to the original code is the RSL communications object, `trans`. The original `main()` routine is virtually untouched, except that the calls to `open()`, `fprintf()`, and `close()` have been replaced with their respective counterparts from the file I/O layer, `c_open()`, `c_fprintf()`, and `c_close()`.

After these minor changes comes the code added by the wrapper. Most of it is *generic* wrapper code, the same for all wrapped modules, but some of it is generated dynamically, according to which routines that the user has specified require new entry points. The first added procedure is the `invalid_request()` procedure. While this could have been a static method in the Message Processing Layer object, we have found that in many cases a user-defined error handler for invalid requests can greatly enhance the ability to debug code and log errors at runtime. Since the `invalid_request()` handler is not a black-box in the runtime system, the ensemble programmer can change it to suit his or her needs. After a program has been wrapped, the ensemble programmer is free to customize it.

The second added routine is the procedure `invoke_other()`. The function of this procedure is to handle *all* message types not defined in the MPL specification. In this example, `invoke_other()` is responsible for the invocation of `wrapped__main()`, `read_in_data()`, and `write_out_data()`. By moving all external invocations (including `wrapped_main()`) to this procedure, the wrapper can wrap codes that do not include an original `main()` and still call routines from those codes. These particular entry points are very simple, as there are no arguments to the original procedures, but the wrapper takes care of casting portions of an RSMessage to procedure arguments. Because the wrapper casts portions of RSMessage directly to procedure arguments, the arguments must always be pre-aligned on word boundaries. Recall that, as discussed in Chapter II, word alignment is a built-in function of the

Remote Service Layer. By using pre-aligned RSMessage and direct argument casts from those messages, we save the creation of local memory space for the arguments and the corresponding `memcpy()`s required to fill those arguments.

The final addition made by the wrapper is the new `main()` routine. This new `main()` is the same for all wrapped MARS modules. Initially, three objects are declared and/or initialized, `trans`, `rt`, and `msg`. These objects correspond to the RSL, Router Table, and Message Processing Layer, respectively.

After these objects are initialized, a call to `change_mode()` is made. If the program is running as part of a MARS ensemble, `change_mode()` does nothing and allows the program to continue. Command line arguments indicate in which mode a program is running. If the program is not being run as part of a MARS ensemble, a call to `wrapped__main()` is made, using the appropriate command-line arguments, and then the program exits. This behavior should be identical to how the original (non-wrapped) code would act.

If the module is not running in stand-alone mode (meaning it made it past the `change_mode()` call), then it immediately attempts a connection to the module that spawned it. This call (`i = trans->rsc()`) is non-blocking; the module may immediately enter its control loop and wait for other messages or incoming connections.

This message loop is no more complex than the loop shown previously (Chapter II), and accomplishes the same tasks. It polls for incoming messages and connections (always as if they are distinct event types, which they *may not* always be) and, if there is such a message, processes the message. This particular instantiation of the control loop has no special exit code, but when the loop exits upon receiving a directive from another module, there are cleanup routines in the Message Processing Layer that are always called. Currently, the wrapper does not accept arbitrary code blocks for exit routines at wrap time, but such support would be easy to add. We have not yet seen a case that requires such unique exit routine support, but it can be added directly by an ensemble programmer.

Once the module has been wrapped, it is ready to be used by itself (as it would have been previously, in stand-alone mode), or it can be used in conjunction with a leader code. For this example, we are going to wrap a second module and build a simple leader. The combined set of modules will allow us to use the output of the second module as a conditional variable and demonstrate simple conditional execution within MARS.

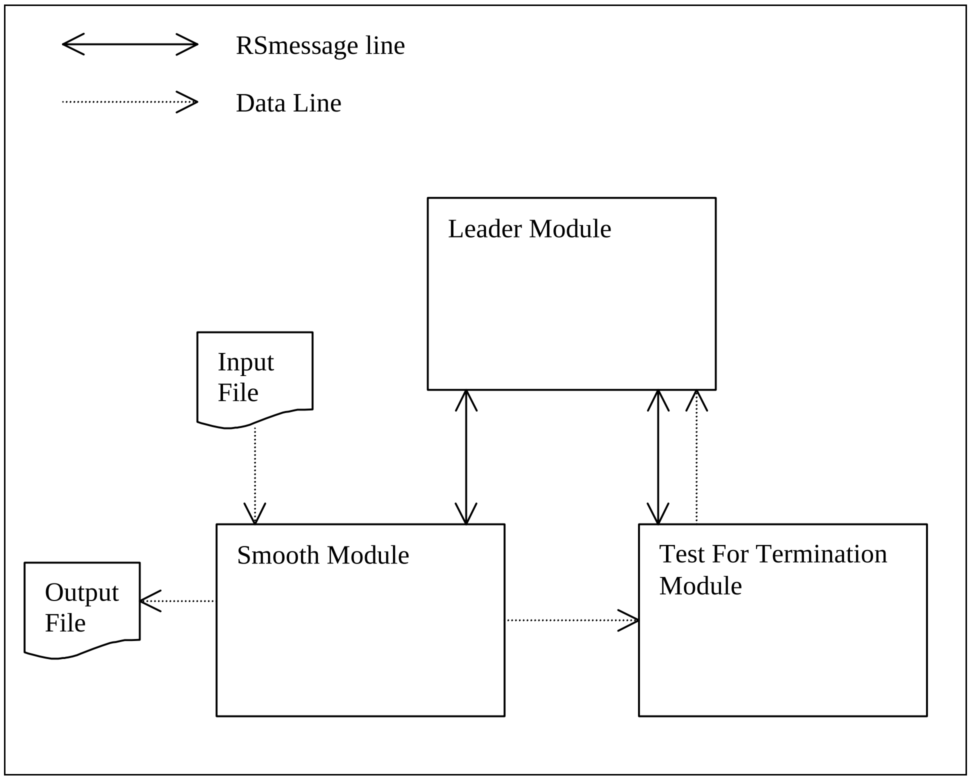

Figure 4. Matrix example

Figure 4 shows what the final module arrangement will be. The leader module will guide the execution flow of the "smooth" and "test for termination" modules. RSMessages are passed along the solid, bi-directional arrows. While these connections are bi-directional, implementers can treat them as unidirectional paths without the need to poll for messages received on the incoming "end" of a connection[13]. Data lines are drawn as unidirectional, though they can be used as bi-directional paths in hand-written modules. However, since these data paths are going to be automatically connected by the MdoModule within the leader code, they will be unidirectional. The leader code will always generate two unidirectional paths to simulate a bi-directional *data* connection. The reason is that if only one side of a bi-directional connection is redirected, it would require significantly more changes to renegotiate connection tables, without the inefficiency of removing and then reconnecting the original connection.

Also, data paths are much more likely to be unidirectional in nature. Even when a program writes to a file and then reads from the same file, there is generally a `close()` and re-`open()` event to reset file pointers.

The leader will spawn the two working modules. As such, there is no need to explicitly create either of the RSMessage lines shown in Figure 4. They are the minimal paths which will always be created during the spawn process. When a spawned module is created, it always "checks-in" with its creator, thus automatically establishing a line of communication with its creator.

[13] A unidirectional connection is a actually a bi-directional connection where one end of the connection is simply ignored.

The data lines seen in Figure 4 are explicitly created by the leader. There are two routines in the smooth module to read and write data to and from disk. Those I/O locations will not be redirected by the leader, but the smooth routine also produces interim output which went to disk in the pre-wrapped code. The leader will set up a data path to direct this output to the “test for termination” module. The test for termination module originally read its input from disk and wrote its output to a file on disk as well. The leader will instruct the test for termination module to receive its input from the smooth module and send its output directly to the leader. Finally, the leader will use the information received from the test for termination module to determine whether to continue execution.

Here is the critical code for test for termination module:

```
int test_for_smoothness(int x, int y, int i, int j,
     int * * data1)
{
  if (((x+i >= 0) && (x+i < MATRIX_SIZE)) &&
      ((y+j >= 0) && (y+j < MATRIX_SIZE))) {
    if (abs((data1[x][y]) - (data1[x+i][y+j])) <=
           SMOOTHNESS)
      return 1;
    return 0;
  }
  return 1;
}

int wrapped_main (void) {
  File * file1, *file2;
  int x, y;
  int * data1[MATRIX_SIZE];
  for ( x = 0; x < MATRIX_SIZE; x++)
    data1[x] = new int[MATRIX_SIZE];
  file1 = copen(“data.set”, “r”);
  for ( x = 0; x < MATRIX_SIZE; x++)
    for ( y = 0; y < MATRIX_SIZE; y++)
      cfscanf(file1,”%d”,&(data1[x][y]));
  cclose (file1);
  int smooth = 1;
  for ( x = 0; (x < MATRIX_SIZE) && (smooth == 1); x++)
    for ( y = 0; (y < MATRIX_SIZE) && (smooth == 1); y++) {
      for (int i = -1; i <= 1; i++)
        for (int j = -1; j <= 1; j++)
          smooth = test_for_smoothness(x, y, i, j,
                      &data1[0]);
    }

  file2 = copen(“result.of.termination.test”, “w”);
  if (smooth == 1)
    cfprintf(file2,”%d “,1);      cflush(file2);
  else
    cfprintf(file2,”%d “,0);      cflush(file2);

  cclose (file2);
  return 1;
}
```

This code originally read the data representing a matrix from disk, tested to see whether it was “smooth enough”, and then wrote its output to another file on disk. The leader can redirect this output because the file I/O routines have been replaced with MARS system primitives.

In summary, this is what is going to happen with these modules:

1. The leader will invoke the *smooth* and *test for termination* modules
2. The *smooth* module will read a matrix from disk
3. The *smooth* module will smooth the matrix and send the matrix to the *test for termination* module
4. The *test for termination* module will check to see if the matrix is “smooth enough”, and send the result to the *leader*
5. The *leader* will repeat from step 3 or proceed to step 6, depending on the result of step 4
6. The *smooth* module will write the final smoothed matrix to disk
7. The *leader* will terminate all modules including itself

To achieve these steps, a leader code is written with the above seven steps in mind. The following code is the actual leader, with the seven steps noted in the comments:

```
rsl * trans;
int
```

```
main (int argc, char ** argv) {
  //setup transport layer. RSL in this case
  rsl * transport = new rsl();
  rss * spawn = new rss();
  trans = transport;

  //setup leader module
  MdoModule * leader;
  leader = new MdoModule(transport, “Leader”);

  //setup “smooth” sub_module
  int smooth = leader->add_module(“smooth”);
  leader->set_host(smooth, “ghengis.cs.uwyo.edu”);
  leader->set_path(smooth,
    “/export/home/mdo/loop/MDO/smooth”);
  leader->set_args(smooth, “”, 0);
  leader->set_file(smooth,(char*)”data.set”, OUT);

  //setup “test_for_termination” sub_module
  int test = leader->add_module(“test”);
  leader->set_host(test, “ghengis.cs.uwyo.edu”);
  leader->set_path(test,
    “/export/home/mdo/loop/MDO/test_for_termination”);
  leader->set_args(test, “”, 0);
  leader->set_file(test,(char*)”data.set”, IN);
  leader->set_file(test,(char*)”data.set”, OUT);

  //connect data paths and start modules  ((STEP 1))
  leader->local_cli_conn(transport->host_name(),
           transport->port_num(), 1,(char*)”data.set”,
           (char*)”data.set”);
  leader->link(“data.set”, “data.set”);
  int status = leader->setup(transport, spawn);
  File * file = copen(“data.set”, r”);
  int done = 0;
  change_mode();

  //read-in data                            ((STEP 2))
  leader->start(transport, smooth, “read_in_data”);
  while (done == 0) {

    //make smooth execute                   ((STEP 3))
    leader->start(transport, smooth);

    //make test_for_termination execute     ((STEP 4))
    leader->start(transport, test);

    //get result of term test here          ((STEP 5))
    cfscanf(file, “%d”, &done);
  }
  cclose(file);
```

```
    //make smooth write out matrix to disk  ((STEP 6))
    leader->start(transport, smooth, “write_out_data”);

    //end all modules                         ((STEP 7))
    status = leader->end(transport);
    return 1;
  }
```

Notice that the seven steps of the process are highlighted in the commented leader code. Once step one is completed, where the modules are setup and data paths are linked, the other six steps in the process are done in 8 total statements (not including the requisite `return` statement). With proper setup, each stage of a process, regardless of complexity, can generally be achieved in about one line[14]. Of course with added dynamism, the leader's complexity grows, but the leader *can* be very simple or very complex, at the user's discretion. In this leader, we use static connection names, because there will only ever be two working modules, and they are always dependent upon each other. We did not take full advantage of MARS' capability to call by modules by name.

Performance is an important consideration with any runtime system. In this simple example, we achieved a considerable decrease in overall execution time, primarily attributable to using network I/O rather than file I/O.

Figure 5 shows the first set of results from the simple matrix example. The platform used for testing consisted of two Silicon Graphics O2 workstations, with R5000 CPUs, running the Irix 6.3 operating system. The machines are connected with 155M/bit

[14] This claim is anecdotal, based on the examples presented here. Of course, ensemble programmers could have more complex routines, requiring many more lines per step.

ATM cards, emulating TCP/IP. Each workstation has sufficient RAM to hold all modules in memory.

Each matrix consisted of 10,000 integer elements, and was generated using the naive `rand()` function, available in most UNIX system libraries. The matrix was smoothed in such away that no two adjacent matrix element values differed by more than 100. Coincidentally, for the initial random seed chosen, the initial matrix required 100 iterations of the smoothing routine before it was “smooth enough”.

Timings were done using the `/bin/time` program available in Irix 6.3. Also, on the other platforms used, including Solaris 2.5.1, the `/bin/time` program was always available, and always the basis of timing. Each program in our test suite had a significant runtime (>10 sec. total time), so the resolution of `/bin/time` was adequate. On the SGI and Linux implementations, `/bin/time` gives two significant digits of accuracy after the decimal, the Solaris implementation gives only one.

For the *original version* of the code, a simple leader was written to coordinate the execution, and calls to `system()` were repeatedly used to invoke the smooth and test for termination modules. This is similar to the methods we encountered with several scientific ensembles tied together by hand. In particular, this was the method used in practice in the scientific code wrapped in the biology model. For the *MARS version*, one single-CPU machine was dedicated to both the leader and the test for termination module, and a second machine was dedicated to the smooth module.

		Run 1	Run 2	Run 3	Average
MARS	real	22.01	21.66	21.55	**21.74**
	user	0.14	0.14	0.13	**0.14**
	sys	0.25	0.25	0.25	**0.25**
ORIGINAL	real	36.35	34.17	32.52	**34.35**
	user	16.31	16.31	16.31	**16.31**
	sys	5.41	5.44	5.46	**5.44**

Figure 5. SGI results from matrix example

Figure 5 shows that our system provided a reasonable increase in performance on two levels (all times are in seconds). First, the total execution time of the original setup was 58% greater than the MARS modules. Second, the overhead of the leader code was drastically reduced as can be seen by the dramatic reductions in user and system time[15]. The method of controlling the program should not be a significant consumer of system resources, which it is in the original example. As can be seen from these results, the relative overhead of our leader code is very low.

		Run 1	Run 2	Run 3	Average
MARS	real	28.4	28.4	28.4	**28.40**
	user	0.1	0.0	0.0	**0.03**
	sys	0.0	0.1	0.0	**0.03**
ORIGINAL	real	55.2	55.3	56.8	**55.77**
	user	27.5	27.6	27.6	**27.57**
	sys	4.9	5.1	5.0	**5.00**

Figure 6. Sun results from matrix example

Figure 6 shows the results when the experiments were performed on a set of Sun Ultra 2 workstations. They were also networked with TCP/IP over ATM, similar to the

[15] This table shows the timings of the respective leader codes, in which total time of the leader is the same as the total time of all associated modules.

SGIs. The Suns were running Solaris 2.5.1. Each workstation had sufficient RAM to hold all modules in memory.

In this experiment, the original program setup takes 96% longer than the MARS modules. This overall speed-up is primarily attributable to removing disk reads and writes, only pushing the data to secondary storage at the end of the execution. The decreased CPU usage by the leader code is even better than seen on the SGI platform.

It is important to note that no further optimizations were performed on the Sun platform (beyond those done on the SGIs). On all platforms used for testing, identical code libraries and modules were used, with no compiler optimizations turned on. The standard MIPS and Sun compilers ("CC") were used throughout, to compile the runtime modules and all test cases.

This example shows how the MARS system is used, starting with source code for a simple module, through the construction of the leader module. Significant performance gains are evident, even without altering the algorithms of the working modules or their parallelism. Also, this setup shows how effective conditional execution can be achieved when tying multiple modules together.

When the user takes advantage of inherent parallelism and leverages that parallelism with the MARS system, even greater gains are possible. We demonstrate this potential in the next two sections. Also, in the third example, the extended pipeline parallelism example, we show the true interoperability of modules in a heterogeneous execution environment.

Pipeline Parallelism Example

This example will show how the MARS system was used to wrap three similar codes into an efficient parallel image processing system. We will use three graphics filters and turn them into MARS modules. As noted earlier in this chapter, the three filters increase sharpness (*is*), posterize (*po*), and remove noise (*rn*) from 8-bit bitmap images.

This experiment uses 40 8-bit black and white images, each 256x256 pixels in size. The images are fed in succession to the three filters. The three filter modules are combined to achieve image compression by bit reduction. The first module, *is*, is applied to refine an images' edges. The second module, *po*, reduces the images' bit depth, using an un-weighted naive heuristic. The third module, *rn*, reduces excessive noise by looking at disparity in neighboring byte values, produced by the bit reduction step.

These three modules are composed with a leader code to produce the ensemble shown in Figure 7.

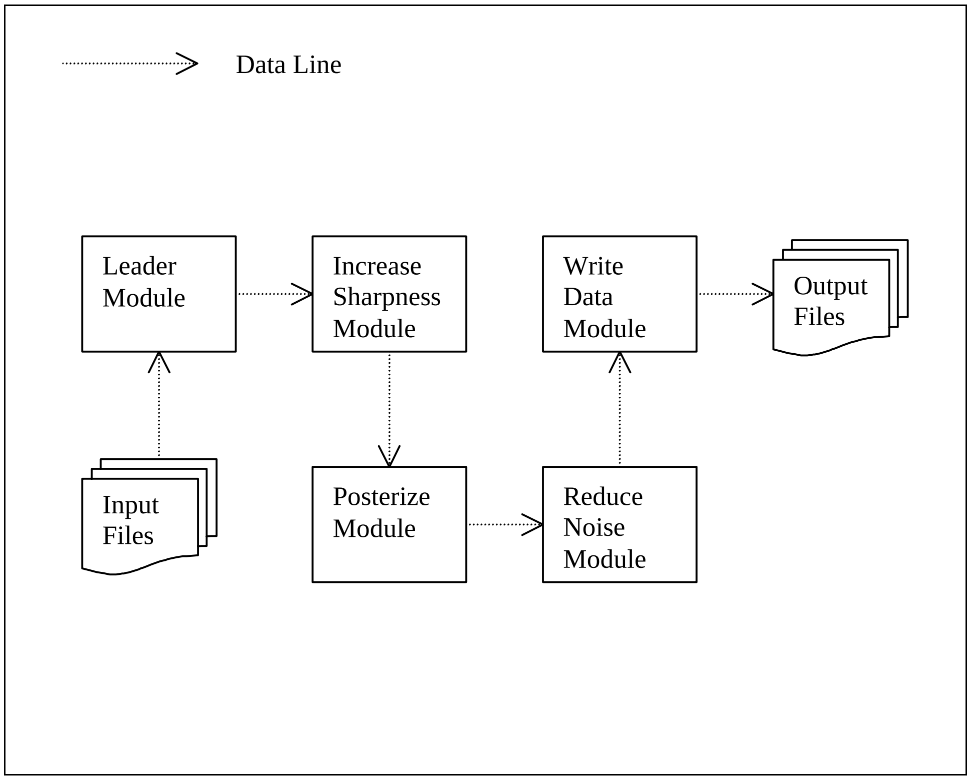

Figure 7. Pipeline parallelism example

The only message lines in this collection are those automatically created by the leader when the modules check in and are omitted from Figure 7. Since these modules do not send control messages to each other, no additional communication lines are necessary.

In this example, we add a fourth module, one written to handle the disk I/O at the end of the execution path. It would be simple enough to allow the reduce noise module to write the images to disk, but this additional helper module's function will become apparent in the next example in this chapter.

The following figures show each stage of the transformation of one of the original images passed through this MARS module collection, as the three filters are applied to it.

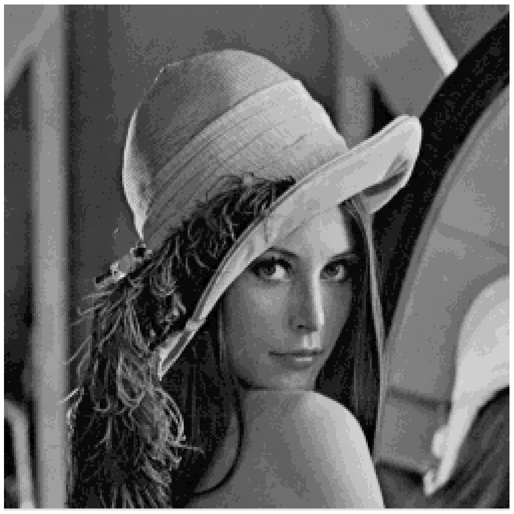

Figure 8. Original image

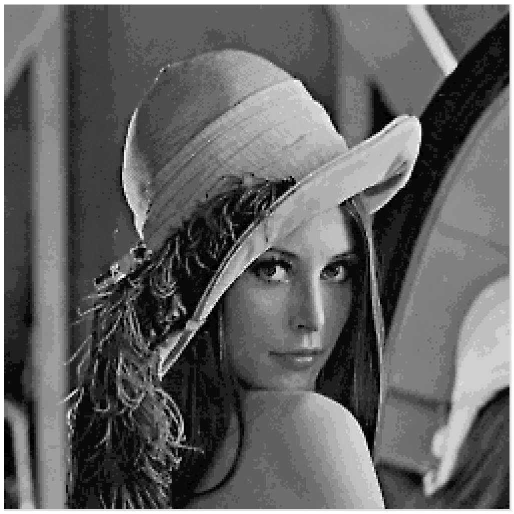

Figure 9. Increased Sharpness

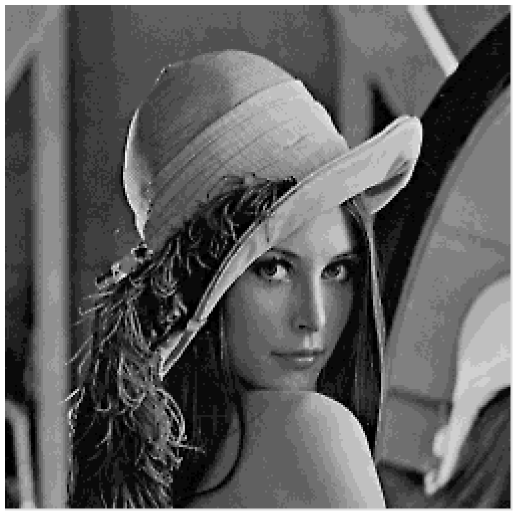

Figure 10. Posterized image

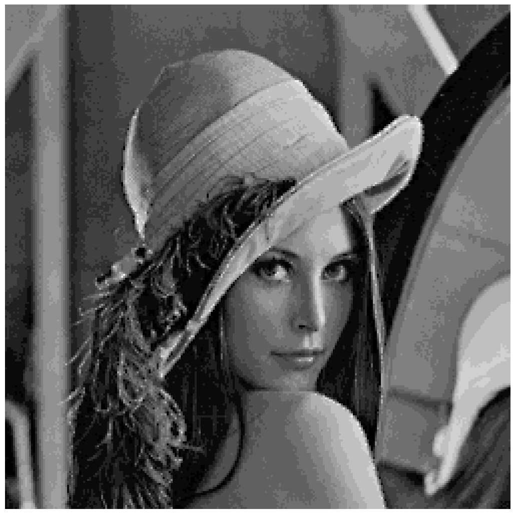

Figure 11. Reduced noise

Figure 8 shows the original image. Changes to the image are often subtle and non-obvious. To best see the changes from Figure 8 to Figure 9, look at the feather in the subject's hat. There is new delineation among the individual strands of the boa. To see the posterization effect, where the bit-depth of the image is reduced, look at the subject's left cheek. There are several distinct bands of color, where there was a previously smooth color gradient. The final changes are also difficult to see, though the feathers again provide clues. From Figure 10 to Figure 11, the feathers are once again *less* distinct, as they were in the original image. The image differences may be difficult to spot, but the purpose of the collected filters is to reduce the bit-depth and resulting size of the image without a noticeable degradation in image quality.

The following code is the leader written to control these four modules:

```
int main (int argc, char ** argv) {
  //setup leader module
  MdoModule * leader = new MdoModule(transport, “Leader”);
  //setup “is” sub_module
  is = leader->add_module(“is”);
  leader->set_host(is, “rodmanf”);
  leader->set_path(is,
     “/export/home/mdo/mdo/tests/II/mdo/sunis”);
  leader->set_args(is, “”, 0);
  isi = leader->set_file(is,(char*)”graphic.dat”, IN);
  iso = leader->set_file(is,(char*)”graphico.dat”, OUT);
  //setup “po” sub_module
  po = leader->add_module(“po”);
  leader->set_host(po, “kublaf”);
  leader->set_path(po,
     “/export/home/mdo/mdo/tests/II/mdo/sunpo”);
  leader->set_args(po, “”, 0);
  poi = leader->set_file(po,(char*)”graphic.dat”, IN);
  poo = leader->set_file(po,(char*)”graphico.dat”, OUT);
  //setup “rn” sub_module
  rn = leader->add_module(“rn”);
  leader->set_host(rn, “attilaf”);
  leader->set_path(rn,
     “/export/home/mdo/mdo/tests/II/mdo/sunrn”);
  leader->set_args(rn, “”, 0);
  rni = leader->set_file(rn,(char*)”graphic.dat”, IN);
  rno = leader->set_file(rn,(char*)”graphico.dat”, OUT);
  //setup “wr” sub_module
  wr = leader->add_module(“wr”);
  leader->set_host(wr, “kublaf”);
  leader->set_path(wr,
     “/export/home/mdo/mdo/tests/II/mdo/sunwr”);
  leader->set_args(wr, “”, 0);
  wri = leader->set_file(wr,(char*)”graphic.dat”, IN);
  //start clients
  leader->local_ser_conn(transport->host_name(),
  transport->port_num(),1,(char*)”graphic.dat”, isi);
  leader->link(iso, poi);
  leader->link(poo, rni);
  leader->link(rno, wri);
```

```
    status = leader->setup(transport, spawn);

    FILE * f;
    FileType * l = new FileType(trans);
    unsigned char input;

    char * file = new char[50];
    strcpy(file, “graphics/g00.dat”);

    for (int done = 1; done <= 40; done ++) {
      f = fopen(file, “r”);
      l->c_fopen(“graphic.dat”, C_WRITE);
      //execute all modules
      leader->start(transport, is);
      leader->start(transport, po);
      leader->start(transport, rn);
      leader->start(transport, wr);

      //read-in data and send to clients
      for (int x = 0; x < GR_SIZE; x++)
        for (int y = 0; y < GR_SIZE; y++) {
          fread(&input, 1, 1,f);
          l->c_fwrite(&input, 1);
        }
      fclose(f);
      if (done % 10 == 0) {
        file[10]++;
        file[11] = ‘0’;
      }
      else
        file[11]++;
      l->c_fclose();
    }

    //synchronous end
    int status = leader->end_reply(transport);
    return 1;
  }
```

Most of this leader code, once again, is devoted to *locating* the working modules on the network. A significant portion of code is also used to read data into the leader. As such, the leader has become an active participant in this collection of modules. Better performance could be achieved if the increase sharpness module read in the data, as it would not have to be passed from the leader to that module. We chose not do this, as the leader would have had to send forty[16] messages to rename I/O connections to the increase sharpness module, significantly increasing the complexity of the collection, and diminishing potential performance gains. Since the data files are only 65536 bytes (64k), the overhead of having the leader pass the data an extra time is not too significant.

Also notice the last call the leader makes, `leader->end_reply()`. This is a synchronous routine. This is different from the call in the matrix smoothing example. In the matrix smoothing leader, the sub-modules were terminated with an asynchronous `leader->end()` call. In the matrix smoothing example, the leader terminates as soon as its job is done, not waiting for the client modules to terminate (though they will do so). The `leader->end_reply()` waits for all client modules to terminate before the leader exits. As such, we could increase performance by using an asynchronous `leader->end()` call, allowing the client modules to terminate on their own time. To make the comparisons to the original filters fair, we wait until the final module has committed its data to disk, ensuring that all files are stable.

		Run 1	**Run 2**	**Run 3**	**Average**
MARS	real	35.6	35.5	35.9	**35.67**
	user	5.1	4.9	5.1	**5.03**
	sys	1.2	1.3	1.2	**1.23**
ORIGINAL	real	69.4	69.0	69.5	**69.30**
	user	50.9	50.3	50.3	**50.50**
	sys	3.2	3.5	3.4	**3.37**

Figure 12. Pipeline parallelism results

[16] One for each image processed.

This experiment was run on the Sun testbed. Figure 12 shows the execution times of the original setup and the MARS modules. Results are similar to the matrix smoothing setup. Once again, the original setup takes about 94% longer than the MARS setup. The system and user times are proportionately longer than in the matrix example, but that is expected since this leader is an active participant in this setup, reading the data files from disk.

With a three stage pipeline, a two-thirds reduction in time would be expected. In this case there was only a one-half reduction in time. With a bit of module profiling, the reason became clear. The increase sharpness module and the reduce noise module take significantly more computation time than the posterizing module. Since those two stages dominate the overall process time, a simple pipeline could only hope to double performance.

With the groundwork for this set of modules in place, we will now demonstrate additional features of MARS that facilitate extending existing setups.

Extended Pipeline Parallelism Example

After improving the performance of the original graphics modules by pipelining and then profiling the resulting collection, it became obvious that a different setup was required for optimal performance. Since the first and third stages (*is* and *rn*) of the overall process dominated the execution times, we multiplied the number of execution nodes at those stages. Figure 13 shows the new module configuration.

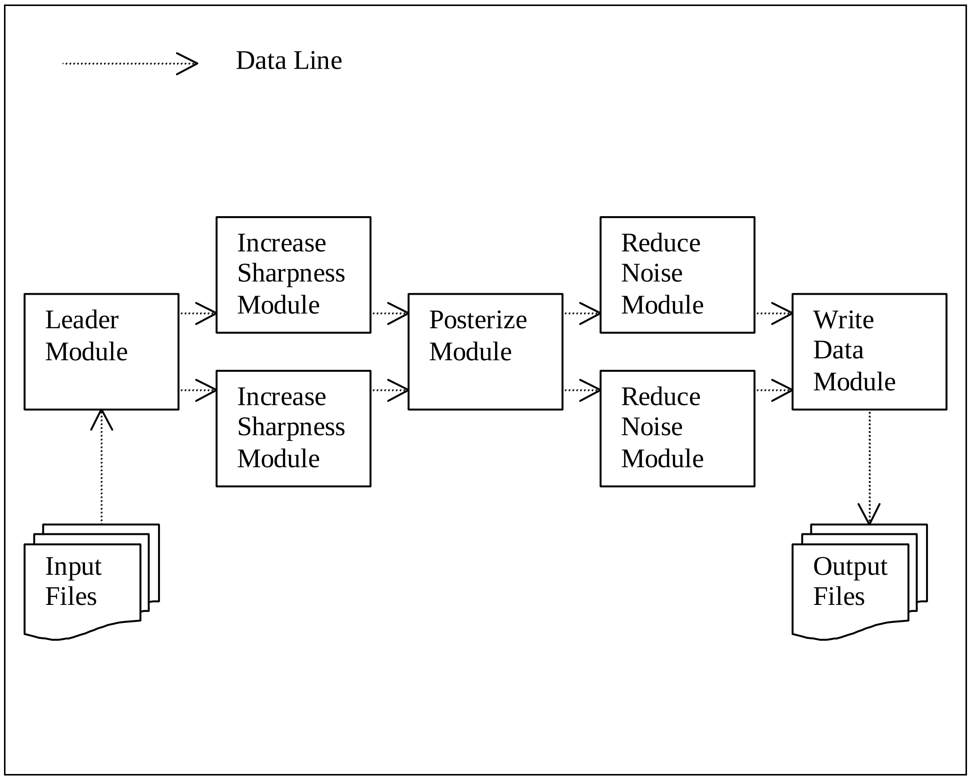

Figure 13. Partial 2-way pipeline

With MARS, moving from the arrangement in Figure 7 to the arrangement in Figure 13 is simple. The leader code remains virtually unchanged, except for the actual control loop. Two new module (one each, *is* and *rn*) specifications are added to the leader, and the leader also adds the required data paths. The control loop in the leader is almost identical, with the data read-in passed alternately to the two increase sharpness modules. The posterize module alternates data to the two noise reductions modules in the same way:

```
for (int done = 1; done <= 40; done ++) {
  f = fopen(file, “r”);
  l->c_fopen(fn, C_WRITE);
  leader->start(transport, isid[is]); //index added here
  leader->start(transport, po);
  leader->start(transport, rnid[rn]); //index added here
  leader->start(transport, wr);
  for (int x = 0; x < GR_SIZE; x++)
```

```
          for (int y = 0; y < GR_SIZE; y++) {
            fread(&input, 1, 1, f);
            l->c_fwrite(&input, 1);
          }
        fclose(f);
        //use the indexes to rename outputs here
        if (done % 2 == 0) {
          fn[7] = ‘1’;
          is = rn = 0;
        }
        else {
          fn[7]++;
          is++;
          rn++;
        }

        if (done % 10 == 0) {
          file[10]++;
          file[11] = ‘0’;
        }
        else
          file[11]++;
        l->c_fclose();
      }
```

The comments in the above code show *the only changes* to the control loop. Two simple indexes are added to determine which parallel modules to execute. The other code fragment (indicated with a comment) toggles both the output path and which modules to execute. Those are the *only* changes made.

To execute the new MARS module collection efficiently, more CPUs were required. To handle the extra load, the leader and write-out modules were placed on the SGIs used in the matrix smoothing example, while the working modules were placed on the Sun execution nodes. This experiment also demonstrates execution using heterogeneous platforms. The results are summarized in Figure 14.

		Run 1	Run 2	Run 3	Average
MARS	real	23.65	24.06	23.12	**23.61**
	user	4.39	4.39	4.41	**4.40**
	sys	2.39	2.23	2.26	**2.29**
ORIGINAL	real	69.4	69.0	69.5	**69.30**
	user	50.9	50.3	50.3	**50.50**
	sys	3.2	3.5	3.4	**3.37**

Figure 14. Partial 2-way pipeline results

Figure 14 shows the results of adding two more modules to the collection. The original setup now takes 194% *longer* than the MARS module collection, or about three times as long. The new setup is faster than the simple pipeline, 23.61s versus 35.67s, a savings of about 12.06s, or faster by just more than one-third of the simple pipeline's total time. Finally, the two-stage pipeline performance about is three times better than the original setup.

This stage-depth parallelism can be arbitrarily extended at any point. We extend the example by adding another *is* and another *rn* module. The leader code requires new information to *locate* the new modules on the network, but really nothing more. The control loop in the leader is changed by only *one character*. In the loop, the fragment `if (done % 2 == 0)` is changed to `if (done % 3 == 0)` to indicate that there are three copies of the parallel modules, rather than two. That is the only change. The resulting configuration is shown in Figure 15.

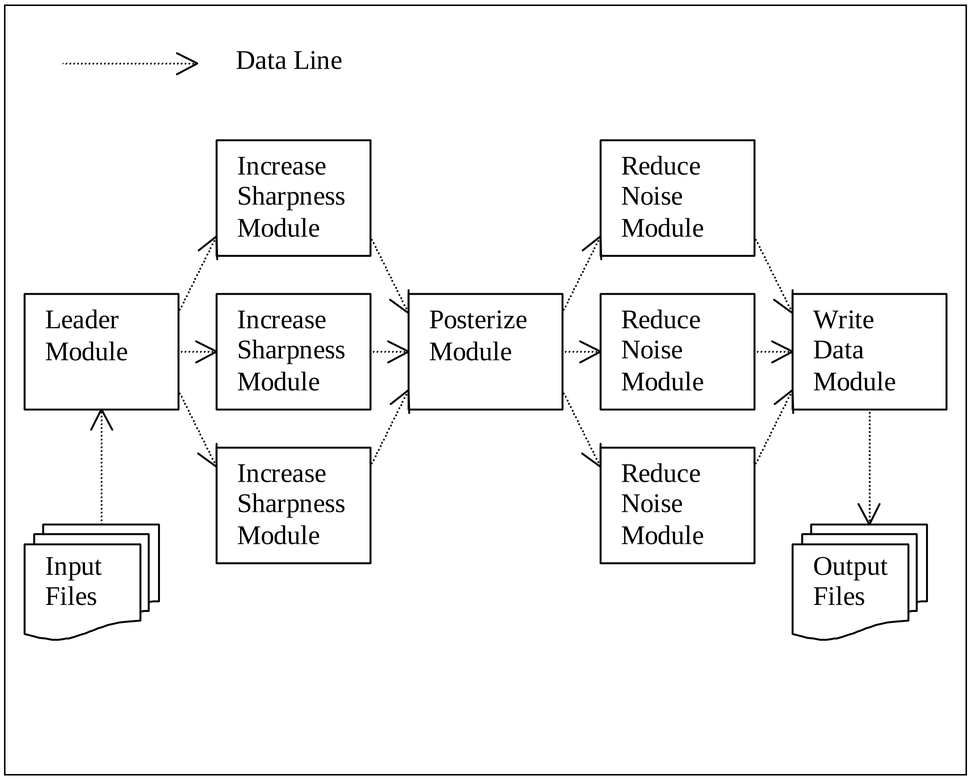

Figure 15. Partial 3-way pipeline

As expected, the execution time is again decreased. The results from the partial three-way pipeline are shown in Figure 16.

		Run 1	Run 2	Run 3	Average
MARS	real	19.02	18.85	19.03	**18.97**
	user	4.39	4.42	4.44	**4.42**
	sys	2.63	2.51	2.53	**2.56**
ORIGINAL	real	69.4	69.0	69.5	**69.30**
	user	50.9	50.3	50.3	**50.50**
	sys	3.2	3.5	3.4	**3.37**

Figure 16. Partial 3-way pipeline results

As before, the runtime is reduced. The amount of time reduction by adding this extra set of modules is not as large as the reduction from the first extra set added. (The original code still takes *265% longer* than the MARS code) A single leader reading inputs cannot keep the pipeline completely filled, nor can a single data writing module commit the data to disk fast enough to scale further, without adding copies of *po* and the write-out module. Three *is* and *rn* stages are all this pipeline can support. Of course, if there were more data readers and writers, and if enough execution nodes were available, the problem could be easily scaled to any arbitrary level, with minimal changes to the leader code. This flexibility allows various scalable problems to be mapped onto numerous configurations of processors with minimal changes to module codes.

This example has demonstrated several more elements of the MARS design. First, MARS has been built to make scaling problems simple. Once a problem has been broken into its parallel components, arbitrary scaling at each stage of the pipeline is a relatively easy process. Also, the partial two- and three-stage pipelines show the support for heterogeneity in the MARS system. The modules in those examples were executed on different platforms, while data and control messages required no alteration. Since the data in the examples was binary, no conversion between platforms was required[17] either.

Biology Model

Description

The MARS system test would not be complete without applying it to an actual scientific MDO code. Each of the codes tied together by hand is unique, and handwritten codes using similar modules vary widely in implementation method and performance.

[17] In the matrix smoothing example, the matrix values were integers. The SGIs and Suns have like byte-ordering (endian-ness) for type `int`, so no data conversion was necessary.

The system we chose to test MARS has three primary modules, including a handwritten leader-like code [Hunt96].

A complete description of the code we applied the MARS system to can be found in “Global net carbon exchange and intra-annual atmospheric CO_2 concentrations predicted by an ecosystem process model and three-dimensional atmospheric transport model” [Hunt96]. This is the abstract from that document:

> A generalized terrestrial ecosystem process model, BIOME-BGC (for BIOME BioGeoChemical Cycles), was used to simulate the global fluxes of CO_2 resulting from photosynthesis, autotrophic respiration, and heterotrophic respiration. Daily meteorological data for the year 1987, gridded to 1° by 1°, were used to drive the model simulations. From the maximum value of the normalized difference vegetation index (NDVI) for 1987, the leaf area index for each grid cell was computed. Global NPP was estimated to be 52 Pg C, and global R_h was estimated to be 66 Pg C. Model predictions of the stable carbon isotropic ratio $^{13}C/^{12}C$ for C_3 and C_4 vegetation were in accord with values published in the literature, suggesting that our computations of total net photosynthesis, and thus NPP, are more reliable than R_h. For each grid cell, daily R_h was adjusted so that the annual total was equal to annual NPP, and the resulting net carbon fluxes were used as inputs to a three-dimensional atmospheric transport model (TM2) using wind data from 1987. We compared the spatial and seasonal patterns of NPP with a diagnostic NDVI model, where NPP was derived from biweekly NDVI data and R_h was tuned to fit atmospheric CO_2 concentrations for 20° to 55° N, the zone in which the most complete data on ecosystem processes and meteorological input data are available. However, in the tropics and high northern latitudes, disagreements between simulated and measured CO_2 concentrations indicated areas where the model could be improved. We present here a methodology by which terrestrial ecosystem models can be tested globally, not by comparisons to homogeneous-plot data, but by seasonal and spatial consistency with a diagnostic NDVI model and atmospheric CO_2 observations.

In the original system, the leader was known as `“gessys”`. The `gessys` module served the same role as a leader module in a MARS ensemble. The other two working modules were labeled `climnew` and `globebgc`. Some of the data transfers and module invocations performed by the `gessys` code were achieved with the following fragment:

```
    system(“./climnew”);
```

```
system("./globebgc");
system("cat grid.day >> newout.day");
system("cat grid.grw >> newout.grw");
system("rm -f grid.mtc");
system("rm -f grid.clm");
```

The three modules were wrapped and tied together into a MARS ensemble. The ensemble is shown in Figure 17. The disk-based output was left in the `globebgc` module. The data files generated for a single run were very large (over seventy megabytes) and took a significant amount of the total runtime to write out. The output was left on secondary (disk) storage in the MARS implementation because it was also available after the original system ran.

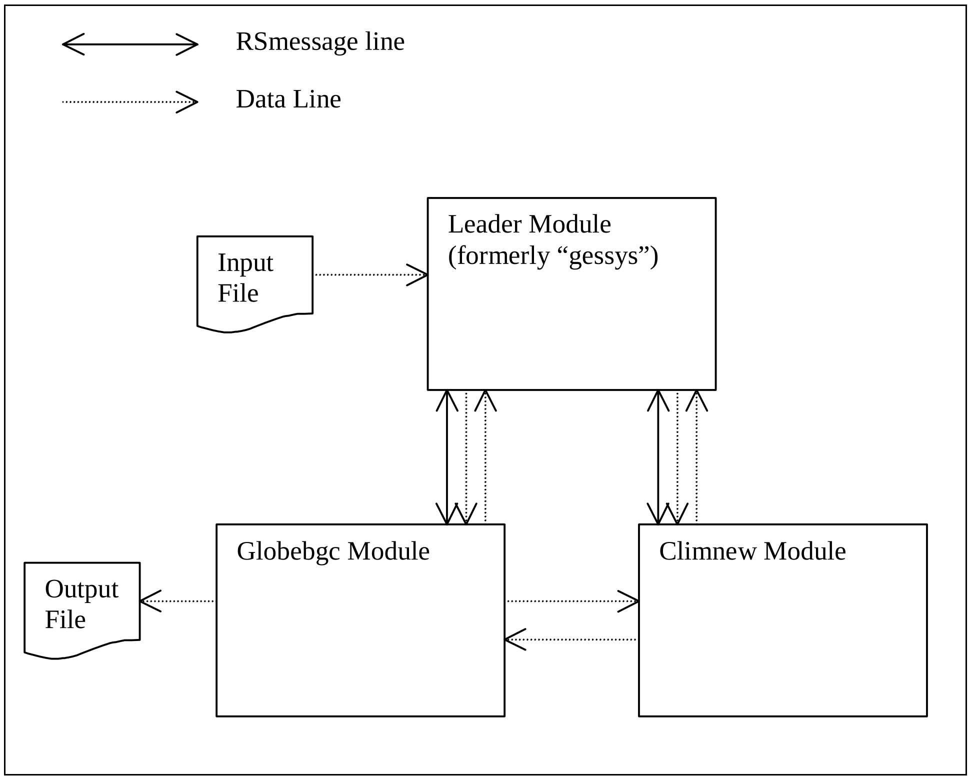

Figure 17. Biology module arrangement

Implementation Problems

The was a series of problems discovered when wrapping the biology code. In overcoming them, we extended the functionality of MARS and learned ways to overcome similar problems with other legacy MDO codes.

The first problem was that one of the modules was written in Pascal. The MARS system does not understand Pascal source code[18]. As such, a freely available conversion utility, `p2c`, was used to convert the module into C source code. The conversion to C source code was not entirely successful after the automatic conversion process, but was close. As such, little additional work was required to get the output from the C-based version to agree with the output from the Pascal version. After conversion to C, the compiled module code was already faster than the original Pascal version. To correct for this compiler advantage, the timings presented for the "original" MDO code are actually the timings from the code compiled *after* conversion to C. This corrects for any differences due to language or compiler. The same compiler was used for the disk-based and MARS-based test cases.

The second problem we encountered involved files that were read multiple times. This was especially problematic if an output file, previously sent to disk, needed to be read in more than once by a second module (e.g., one file had to be read twice after being committed to disk). There are two obvious solutions to the problem: buffer the file locally and send it a second time, or send its data twice, over two different data lines. The reason the data would have to be sent over two different virtual data connections is

[18] See the chapter on future work for further discussion of the language problem.

that the data needs to be both ordered and complete. Given the ease with which connections can be dynamically renamed, sending data out over two lines does not present a problem to the module reading in the data. Rather than developing a new method to buffer a potentially large file locally and then re-sending it, we chose to use two connections and to write the data two times, without local buffering. The biology module read a very small file twice, so this was an appropriate solution. In the original code, the file `grid.dat` was read in twice and output twice as well.

Of course, there is no reason for a stable file must be read twice by the same program. The data can always be stored in local variables, or the file can be stored in memory and accessed a second time without looking to disk. There are times when writing a second copy of a large file to the network can be a problem, especially when the second output stream is not accessed until the first is completely consumed. This can cause blocking to occur in the data writer when system buffers on the receiver's end are full. As a work-around to that problem, the data buffer mentioned in Chapter III was developed. It allows files of arbitrary size to be buffered online, preventing modules writing to the network from blocking while waiting for their data to be consumed.

A third problem also involved the `grid.dat` file. In addition to being read twice by the `globebgc` program, it was read a third time by the `climnew` module. The solution to this problem was simple; a third copy of the file was output to the network, with a new destination. The real problem this presents is that the way these virtual files are treated by the network are different from how they are treated on disk. Once a file goes to disk, it can be used as if it were *stable*; it does not have to be recreated whenever it is needed. It may be accessed one time, perhaps a multiple times, or perhaps never. At wrap time, the wrapper cannot “know” that an output file will be accessed multiple times or by multiple modules. The programmer is responsible for this consideration. We cannot foresee any automated solution to this problem without simply implementing a true file system over the network, which would require stable storage (in the form of disks) to store all the potential information (thus defeating the purpose of the network abstraction). Since each MARS ensemble *does not* require a complete file system, some analysis by the programmer is warranted to work around this multiple-file-open problem, thus preserving the system performance.

The fourth problem we encountered was flushing output at the proper time. If data was flushed too often, the amount packet and header information dominated amount of actual data, and the network was unnecessarily inundated with excess information. If the data was flushed too rarely, modules depending on particular data could stall, slowing (or deadlocking) the entire MARS collection. Data written to the network using the file I/O layer is actually buffered locally first, and *eventually* written to the network. Currently, since there is no threading in the I/O handlers, it is not possible to flush data except when an explicit call to a file I/O routine is made. Data is flushed at three points: when the local data buffer is full, when an explicit call to `c_flush()` is made, and when a virtual file is closed. The problem we encountered is that receiving modules can be left waiting for essential data, especially when two modules pass small amounts of data back and forth, and the modules are mutually dependent on each other’s data. To solve this problem, we manually added `c_flush()` calls to the biology code after large blocks and loops which contained output statements. It was found that, in almost every case, these loops and blocks represented strongly correlated data elements and that flushes solved dependency problems. In most codes, flushing should not be a significant problem, as it is unlikely that independently developed modules would have mutual dependencies.

The mutual data dependencies were created because these modules were intended to operate as an integrated system, and dependencies were created when they were originally tied together by hand.

The fifth problem we had was with the I/O done as `printf()` and `scanf()` calls. Virtual `printf()`s are available by default, and virtual `scanf()`s can be easily constructed by using the `stdargs` or `varargs` library routines. On the other hand, some things are still very difficult to do. For instance, performing a `scanf()` to read an entire line, when only half the line has been received over the network, causes catastrophic problems. Since network transmissions occur in packets with size determined by TCP/IP, perhaps only half of a single `printf()` string would be transmitted in one packet. If that packet is read in by the native transport layer, and the program then tries to scan an entire line out of that string, what occurs? Blocking with no chance of return.

To work around this problem, we sent output elements as their binary equivalent, with the size of the argument known at runtime. This required some additional work to guarantee that data would be sent and received properly, but it is a rather simple process to convert an integer written as a string to one written in binary. Of course, for interoperability and consistency, any final output files committed to disk by the original code are still written in the same form by MARS. It is just the data written between modules that has been converted to binary (when it is not already binary).

This conversion of the output to an interim binary form lead to the sixth problem: precision. Since some data was originally written to disk and then re-read from disk, binary sends could lead to differences in output precision. For example, if a type `double` element whose true value was “6.125” was written to disk with only one place of precision, it would be stored as “6.1”. If it was later read from disk into another element of type `double`, its value would certainly not still be “6.125”. On the other hand, if the data were sent between the modules in binary form, the values would be identical on both ends, and no precision would be lost. Rather than waste precious cycles *losing* precision to guarantee that all outputs agreed exactly, we chose to treat the *increased accuracy* as an acceptable side-effect.

Results

The three modules were executed on the SGI and Sun test-beds. They were first run on both test-beds using a limited data set. They were also executed on the SGI test-bed with the full data set (70+ megabytes of data) used in the literature that originally presented the biology model. The results are shown in Figure 18, Figure 19, and Figure 20.

		Run 1	Run 2	Run 3	Average
MARS	real	23.3	23.6	23.9	**23.60**
	user	2.0	2.4	2.5	**2.30**
	sys	0.5	0.7	7.0	**2.73**
ORIGINAL	real	49.0	49.2	48.8	**49.00**
	user	23.1	23.3	22.8	**23.07**
	sys	4.2	3.6	4.0	**3.93**

Figure 18. Small data set on the Suns

		Run 1	Run 2	Run 3	Average
MARS	real	17.49	17.49	17.67	**17.55**
	user	2.26	2.41	2.39	**2.35**
	sys	2.65	2.84	2.82	**2.77**
ORIGINAL	real	39.38	39.07	39.22	**39.22**
	user	15.46	15.44	15.47	**15.46**
	sys	5.13	5.04	5.10	**5.09**

Figure 19. Small data set on the SGIs

		Run 1	Run 2	Run 3	Average
MARS	real	5833.55	5578.12	5783.15	**5731.61**
	user	80.68	80.91	81.10	**80.90**
	sys	18.34	18.73	18.36	**18.48**
ORIGINAL	real	8376.64	8407.94		**8392.29**
	user	3476.26	3459.28		**3467.77**
	sys	989.06	984.31		**986.69**

Figure 20. Large data set on the SGIs

The first set of results, shown in Figure 18, summarizes the performance on the Suns with the small data set. The original setup takes 108% longer to run on the Suns than the MARS version of the biology code. This is an excellent time improvement; while only one extra CPU is being used, there is a speedup of over 100%. The original *problem* scales perfectly (1:1) and still goes a bit further. The original *code* could not be readily scaled across two CPUs (even with identical file systems) because there are no synchronization guarantees with UNIX files. One module had to run to completion before the second was executed. That problem was solved with our binary I/O implementation.

Figure 19 shows the same small data set when run on the SGI test-bed. The speedup was greater than the speedup on the Sun test-bed. The original code took 123% longer than the MARS implementation on the SGI machines. The user and system times for the MARS implementation on the SGIs were approximately equal to those on the Suns, but the overall time was considerably less on the SGIs. Of note, the original code ran faster on the SGIs than on the Suns, and the proportional overall speed-up was greater as well.

Figure 20 shows the results from the large data set run on the SGIs. The original code took 46% longer to run than the MARS ensemble. The user and system time speedups were significant when using the large data sets. The user time of the original code was approximately forty-three times MARS' time. The system time was about fifty-three times greater using the original system. The extreme speedup in those areas was expected. The relatively poor showing in total time difference was attributable to the massive stable output files generated by the modules. Generating the large output files actually takes more time in this simulation than the computations.

V. Conclusions

MARS proved to be a successful solution to the problem of integrating multidisciplinary codes. There were many features of MARS that contributed to its overall success. Those primary achievements are outlined in this chapter.

Improved times

On each the test cases in Chapter IV, there were significant speedups. Overall runtimes were improved in each of the four main examples (matrix multiplication, pipelined graphics filtering, extended pipelined graphics filtering, and the biology code).

In the worst case, the matrix multiplication example, only a single data set was read from disk, minimally processed, and written to disk again. In that instance, the original code took about 60% longer to execute than the MARS code. The explanation for this lackluster performance is simple: the initial disk reads and writes dominate the process time. Where this is the case, little can be done to improve the performance of the code. Many scientific codes, however, will have multiple data runs and/or many more intermediate states that will contribute to better performance with MARS. The worst case program to apply MARS to would be a single module that reads a data file and immediately writes it to disk again. We would expect no performance gain. At the same time, such a program is certainly not *multidisciplinary* in any sense and not the target of MARS. This matrix multiplication program is the absolute minimal case that could be *multidisciplinary* (two modules), and it still shows reasonable performance gains.

In all other cases, performance of the MARS ensembles was very good. The extreme performance gain was shown in Figure 16, the partial three-way graphics pipeline results. The original code took 265% longer than the MARS modules or about 3.65 times the total MARS time. Were we to discount setup overhead and assume already running modules, that time would also be further improved. Also, we timed the ensemble with synchronous module terminations, which can be significant with the nine modules involved (Figure 15). Using an asynchronous termination call could have yielded better results as well.

Overall system load is reduced by using MARS data paths (where possible) instead of file based I/O. By diminishing system and user times, overall throughput of particular execution nodes can be increased. This is an important consideration. If one machine can be used more efficiently, it frees up valuable cycle time for other processes and/or additional MARS modules. For instance, in our three-way pipeline parallelism example, we placed several modules on a single-CPU SGI and saw excellent performance because significantly less time was spent reading, writing, and verifying data previously written to disk. This time can be used for additional MARS modules or for any other processes the user chooses.

One of the main reasons to consider MARS for multidisciplinary codes is performance. A large part of the performance gains come in overall execution time. While overall time can be significantly decreased, user and system time can be reduced as well.

Distribution and realignment

Distribution of modules is very simple with the MARS system. Distribution comes in several forms, including setup methods of parallel codes and various types of dynamic realignments.

The presence of the leader code makes setup of distributed codes very easy. As long as a working module exists on an execution node that supports MARS, the leader can simply indicate that that node will be responsible to run that module. A node that supports MARS calls does not need to be aware of any working modules located on that node. As such, adding working modules to a MARS node is as simple as making an appropriate binary available to that node. While there is no current mechanism to install working modules on a MARS execution node remotely, such an enhancement is not difficult to envision.

On a second level, altering parallel codes is easy. As shown by the two-way and three-way pipeline parallelism examples, changing a single constant value can be enough to expand a parallel stage. With proper implementation, such expansions (or contractions) occur with little or no change to any module, save the leader.

Third, realignment of codes is an easy process. Switching from one set of execution nodes to another is not difficult. Changes are only required in the leader module. Such changes can be little more than single line changes, indicating the DNS name of the node to host a particular working module (the use of DNS assumes TCP/IP communications). This realignment can occur dynamically as well. Properly constructed leaders can arbitrarily add or remove modules from a collection during execution. It is not difficult to imagine when this type of dynamism could be important. Given a defined set of MARS nodes, the leader code could expand a collection to consume the maximum allowable amount of resources. The leader could similarly contract a collection of modules when critical resources are required by other users/processes.

A final advantage of the MARS system is the late binding time for working module descriptions. In reality, a completely generic leader code does not require *any* information about the modules that it is going to execute at compilation time. The complete specification for a collection of working modules could be input from a file at runtime, or even from command line user prompts. In most implementations, the leader has module locations hard-coded into it because execution nodes (especially in a laboratory environment) are often static. Runtime assignment is very possible, and experiments show that it works well with each of the examples shown in Chapter IV[19].

There are four distribution and realignment advantages of the MARS system. The first is that leader modules can be changed slightly and effect significant changes in the overall setup of working modules. The second advantage is that altering parallel codes is a simple process. The depth of particular stages of the ensemble can be changed from the leader code without altering the working modules. The third advantage leverages off the first two: at runtime, sophisticated leaders can add and remove modules from running ensembles. The fourth advantage is that the late binding time for module identifications

[19] Times for such experiments were not included. The reason is that total runtime performance suffered drastically when the time required for a user to *type module specifications* was significant compared to total runtime. It does not seem fair to compare a static hard-coded system to a MARS module crippled by a user who types slowly.

means that leaders can be ignorant of even their working modules until runtime, and the working modules can even be specified from the command line by users.

Heterogeneity

Because data and control messages are passed using an abstract type, MARS implementations can run on multiple platforms without requiring platform-specific messaging code. Of course, platform specific *optimizations* are possible, but do not affect compatibility. The RSMessage type is composed of byte (`char`) fields that are dynamically allocated and word-aligned based on the data types they represent. As such, network transmission and message translation are simple processes.

Since most other MARS functions are based on standard protocols, there has been no problem porting the MARS runtime system to many different platforms. The current runtime system compiles on Sparc, Intel, and MIPS CPUs running Solaris, Irix, and Linux *without any conditional compilation statements*. The current implementation of the runtime system has been written so that there are absolutely no `#ifdef #endif` pairs that are dependent upon platform or OS specifics. In addition, the only `#ifdef` required is a *compiler* dependent type definition, specifically that the newest versions of gcc/g++ recognize type `bool` as a primitive type but the MIPS CC compiler does not. Perhaps the term “porting” implies too much. MARS compiles *without changes* on all mentioned UNIX implementations.

Finally, since the protocols used are standard, even machines that lacked particular services to support MARS were easily augmented to support the system. For instance, ATM can easily (and freely) emulate TCP/IP. Of course, it is not as fast as native ATM, but the performance gains using emulated TCP/IP were still significant. Perhaps future MARS implementations will include ATM support at the native transport layer level. On another front, SGI's Irix OS has internal support for the TCP/mux protocol but Solaris 2.5.1 does not. (Recall, the TCP/mux protocol was used to spawn programs on a MARS node.) It was an easy process to implement RFC 1078 for the TCP/mux, and the implementation has been included in the MARS distribution [Lottor88].

MARS has been compiled and run and many different flavors of UNIX and several popular platforms. MARS modules concurrently running on different combinations of OSes and platforms readily cooperate and exchange data and control messages without difficulty.

Conditional execution

With the addition of modules designed to return control variables, conditional execution can be achieved with MARS. Simple conditional execution was demonstrated with the matrix smoothing example.

By augmenting an ensemble of working modules with *testing* modules, the ensemble can be terminated early or runtime can be extended as needed. In addition, certain modules can be executed (or *not* executed) as needed, based on the outputs of testing modules. For instance, imagine an iterative approximation process (like Newton's method) designed to calculate a numerical result. With poor starting values, such processes can quickly diverge. It is of great benefit to be able to detect such divergence quickly and restart the process with different initial values. By the same token, a converging process that has not yet achieved a certain level of precision should be extended rather than terminated and subsequently restarted with an increased number of iterations.

Such conditional executions are possible with MARS. By adding modules to inspect process data at arbitrary checkpoints or by having the leader inspect the data, arbitrary executions may be branched to achieve maximum performance.

Authoring system

As an added benefit, we have found the MARS system to be an excellent authoring support system for parallel and distributed codes. The initial impetus for MARS design was to wrap existing multidisciplinary codes into MARS modules that could cooperate on large problems. After early success with the initial MARS problem area, there were indications that the MARS system was appropriate for the design of new codes, not just wrapping existing codes.

Data transmission between modules is quick with MARS, as is the transmission of arbitrary messages. Wrapped modules can be spawned on any node that understands the TCP/mux request for a module. Modules can be easily inserted or removed from ensembles. Anywhere that dynamism is important or where an easy to use, richly featured communication and control system is desired, MARS can be applied. Currently, two other M.S. in Computer Science theses at the University of Wyoming (both in progress) depend on early implementations of the MARS system. Those projects are not related to the development of MARS, but use MARS in novel systems created from the ground up. One system uses MARS to effect a distributed and network aware version of the Revision Control System (RCS); the other uses MARS communication and spawning to implement a Geographical Information System (GIS) search tool composed of multiple modules and database interfaces.

New parallel and distributed codes are easily built with MARS primitives. While this was not the initial design impetus of MARS, the potential is there and will be explored in the coming years.

VI. Future Work

While designing and working with the MARS system, several additional components were suggested. Some of these modifications have clear implementation paths, while others are still open ended problems. With these proposed features in place, MARS would be one of the most fully functional distributed runtime support system available. With the support for integrating existing multidisciplinary codes, these enhancements effectively remove any limits to designing new applications or modifying existing ones.

Dynamic scheduling

As observed in Chapter V, the message layers and dynamic invocation schemes allow for dynamic scheduling with properly constructed modules and leaders. Actually constructing such leaders and modules took significant experimentation to develop generic extensions to a module that would facilitate dynamic scheduling. The code to effect the dynamism is still somewhat limited.

The dynamic scheduler we propose would integrate all the necessary information in special structures contained in a leader code and would include standard stubs in working modules. The scheduler would have two main components, the *load balancer* and *execution sets*.

There would be an execution set for each working module in a MARS collection. For instance, assume an ensemble with three modules, named “A”, “B”, and “C”, with a data flow path from “A” to “B” to “C”. Also, assume the depth of each stage of the ensemble (“A”, “B”, or “C”) is extensible to an arbitrary limit. At the same time, there are exactly ten available execution nodes, “0” .. “9”, each capable of running only certain modules. An *execution set definition* might appear as the following:

```
A.execution_set(0,1,2,3,4,5);
B.execution_set(0,5,6,7,8,9);
C.execution_set(1,2,3,5,7,9);
```

These definitions would appear in the leader module and would not need to be made available at compile time or runtime to particular working modules. (According to this specification, module “A” could be executed on nodes “0” .. “5”.) The leader would start the modules and establish the necessary data connections. The leader may or may not be given a specific initial configuration (e.g., module “A” should be started on nodes “1” and “2”, etc.). If not given an initial configuration for working module, the leader would simply place modules on whatever nodes are appropriate to approximately balance the depth of each module layer (e.g., module “A” could have three instantiations, module “B” could have four, and module “C” could have 3). After running, a module would be responsible for reporting the local runtime for its `wrapped__main`, and the load balancer would re-distribute modules as necessary to balance stage throughput.

The load balancer would redistribute nodes to balance each overall stage runtime. For example, given two instantiations of module “A”, one running to completion in twenty seconds, the other running to completion in thirty seconds, the overall “A” stage runtime would be twelve seconds. Effectively, one complete run of “A” could be achieved every twelve seconds. It is a simple linear equation to calculate overall stage runtime (e.g., for module “A”, $1/30s + 1/20s = 1/\boldsymbol{x}$, $\boldsymbol{x} = 12s$). If one stage takes significantly longer than another, and if making changes to the module configuration would be likely to yield overall throughput benefits, the load balancer would make such changes.

The stubs located in the working modules would only be responsible for reporting local runtimes, and would only communicate results to the leader modules. Priorities could also be assigned to members of a module's execution set, if prior performance knowledge was available. Also, particular nodes within the execution sets can have limits placed upon them, such as that only one module at a time may reside on a particular node.

While this functionality is possible with the current set of MARS primitives, it currently involves writing the scheduling code (load balancer) by hand in each leader. Rather than writing and rewriting this code every time, it should be added to the functionality of the base leader code. Also, timing data must be reported by working modules for this system to be fully functional. We envision a complete system with support for MARS acting in a way similar to the process planner described in [Adler95].

Multi-language support

Currently, the wrapper only works on codes written in C. Some other languages, such as Java, are similar in both syntax and available primitives. For instance, socket objects for TCP/IP communications in Java are virtually identical to those in C. Java is also uniquely suited to realize the control loop that exists in all working modules. Adding support for Java should not be excessively difficult, but has not yet been done.

Some languages, like High Performance Fortran, are syntactically different from C, but otherwise support the primitives necessary to support this implementation of

MARS. TCP/IP routines, for instance, are currently available to Fortran. A MARS wrapper for Fortran codes is not currently available, though is likely to be developed soon. Many legacy codes are written in Fortran, and the ability to deal with a large available code-base is certainly desirable.

Other languages, like Lisp and Prolog, are very different from C, and would require significant extensions of MARS to support them. For instance, for interpreted Lisp, a whole new spawning system would have to be developed. With Prolog, message passing, even with TCP/IP, is not an easy task. These other languages are generally not employed by multi-module authors, except in heterogeneous implementations and often exist without network support. A cursory examination reveals that the brunt of scientific multidisciplinary codes are written in languages like C/C++ and Fortran (with various extensions), with a growing number of implementations using Java. It is not clear at this time how the wrapper would function with a source written in an interpreted language, such as Lisp.

To support existing codes fully, MARS should be able to wrap source written in multiple languages. In our own examples, we converted Pascal and Fortran routines to C before adding them to ensembles of MARS modules.

Native threading/thread support

Currently, there is no support for threads in this implementation of MARS. Given “friendly” modules with their own threads, MARS treats them like any other non-threaded code and can wrap them without issue. For the most part, any code that has threads spawned by its original main routine and such threads exit gracefully (with the exit from the original main routine) can be turned into a MARS module without a problem.

There are two main reasons to add some thread support to the MARS system. First, for those who would use the MARS system to author new module collections, native thread support would be convenient. It would be better to have a threads package known to be compatible with MARS rather than requiring users to use an alternate implementation, like Pthreads or OpenThreads.

The second reason is that I/O can be significantly improved with threading. Recall that there can be deadlocks in an ensemble if two modules depend on each other's output and that output is not explicitly flushed at certain stages. If the I/O routines were threaded, after reaching a certain time-out, if data was still present in a module's buffer, that data could be automatically flushed to the network or disk. Currently, users are responsible for preventing this type of deadlock by using explicit flushes or closes[20]. It would be easier if the ensemble creators did not have to worry about this type of problem, as they generally do not have to with more traditional codes. For instance, data written to disk will *eventually* get put to disk, without an explicit flush or close by the user. Currently, data written to a MARS data path *may not*, without a flush or close.

Threading allows the user to "forget about" flushing data to the network, and the corresponding headaches such worries can cause. With "native" thread support (even if it simply consists of Pthreads or another implementation), newly created MARS modules

[20] MARS *does* data flushes automatically when an I/O buffer is full, but implementers may have dependencies on data that do not fall on this buffer boundary or are simply smaller than a single buffer size.

can be assuredly safe, without worrying about misbehaving third-party threads running amok.

Another advantage of threading I/O is that when too much data “piles up” on a TCP/IP connection without being consumed, the data’s emitter must block until some portion of the data is consumed. Currently, we have been able to work around such problems with relative ease, by using the (*threaded!*) data buffer module, but *not* having to work around the problem would be a much better state.

A final advantage to threading is that the message processing could continue unabated, even when a `wrapped__main` (or some other routine) was executing. Such message processing could be used to update Router Tables while a routine was being executed and before another output routine started. Of course, enhanced steering mechanisms can be envisioned with threaded message processing. If at mid-execution the output from a thread was no longer required, that thread could be terminated, at a savings to the entire ensemble [Kleiman96].

Automatic data conversions

Currently, there are no automatic data conversions in the MARS system. All data conversion, where necessary, is done by specially designed proxy modules. For example, if 32-bit integer data is sent between machines with unlike endian-ness, one using big endian and the other using little endian representation, each integer must be reversed in a bit-wise manner by a proxy module.

There are several different standards for dealing with heterogeneous data typing, including Abstract Syntax Notation One (ASN.1), X.409, and the Basic Encoding Rules

(BER). No support for any of these protocols is currently available in MARS. We have chosen to examine a unique system, Smartfiles, to address the data exchange problem [Haines95b]. The Smartfiles subsystem takes care of mundane issues such as endianness and array ordering and access, but allows further functionality by encoding information in description fields *about* the data being transmitted. By mapping these fields onto configurations different from an original Smartfile, unique data access patterns can be achieved with relative ease [Haines95b]. Such functionality would be a boon to multidisciplinary codes that pass data to other modules that do not require all the output fields or only require a simple data reordering.

At this point, given the added functionality of Smartfiles, it seems to be a logical choice for automatic data conversion. Not only does it provide a black-box standard for data conversions, it can make binding different modules into MARS ensembles much easier. Even sharing data through a consistent standard like ASN.1, legacy modules are unlikely to have the same data format requirements. The integration of Smartfiles would be an important tool to overcome differences in legacy module data requirements.

Strong profiling support

Basic, coarse-grained profiling is available in MARS. By setting log levels high, every message sent and received or processed can be tracked and stored for later examination. By examining message log time differences, runtimes of blocks of code can be calculated.

The logs do not provide much profiling support, however. It does not approach the information provided by a program like `prof` or `pixie`. At the same time, there have been some problems using standard profilers with particularly complex MARS codes, especially those that wrap MARS routines around threaded codes. The same problems exist for stand-alone threaded codes as well, but native support for profiling would be extremely helpful in troubleshooting and tweaking performance in MARS modules.

With native profiling support, users could get breakdowns such as "how much time is spent on data transmissions," "how much time is spent processing messages," and "how much time is spent in various module routines." Currently, the only profiling is very coarse and therefore not very informative. Better profiling could lead to better modules and better performance.

Better steering mechanisms

There is not much a user can do to steer a MARS collection, other than writing an effective leader that "knows" what effective steering is. Of course, compiling all such knowledge into a single leader code is impossible and probably not worth the effort required.

How does a user terminate an entire collection of MARS modules? Currently, there is no easy way to do so. Terminate the leader, and subsequently terminate all working modules: that is the process. There is an experimental terminator for leader codes that installs a signal handler to receive a SIGUSR1 (or other programmer-specified signals) and then send termination messages to all working modules before the leader terminates. This brute force approach is not at all elegant and does not support anything other than terminating a complete setup.

We envision a signal handler that allows much greater user control over the entire process. For instance, working modules should be able to report their pending messages and their current threads of execution. Execution sets for modules should be capable of dynamic expansion and contraction *at the user's discretion*. Leaders should be able to invoke extra modules *on command*. Users should be able to examine an ensemble's progress by querying leaders or working modules.

There are many different reasons to steer running collections, far beyond simply indicating that an entire collection should terminate. Status information and changes to the entire setup should be possible with a running MARS ensemble. Better steering mechanisms are a priority in the future expansion of MARS.

Better condition variables

Condition variables are not really present within the MARS system. We demonstrated conditional execution by building a module to test a data stream, but this extra module was constructed by hand and did not really leverage any existing MARS structures or primitives.

To achieve better conditional execution, there should be a way to check execution status of working modules while they are running. (To do this first requires thread support in MARS.) Once such support is available, basic methods could be added to MARS to make many types of condition variables automatically available to a user. For instance, the following code would be easy to build with enhanced status information available:

```
leader->start(ModuleA);
```

```
while(! ModuleA.done()) {
  leader->start(ModuleB);
}
```

This code fragment could guide an approximation system, where `ModuleB` constantly refines an answer until `ModuleA` is done, or anything else the reader imagines. Basic condition variables, such as `module.done()`, can be used in many ways, and are not limited to simply checking for execution completion. Real-time systems could also use similar state variables to terminate processes whose answers have expired:

```
leader->start(ModuleA);
for (int x = 0; x < 30; x++, wait(1)); //30 sec runtime
if (ModuleA.running())
  leader->end(ModuleA);
else
  //collect ModuleA's data here
```

Currently, a user could send a message to ModuleA that simply asks for a reply and hope that the message is answered in a timely manner. But what is a *timely manner*? One extra second? Two? Such a method would be messy and unpredictable, at best. The addition of condition variables could greatly enhance the functionality of MARS.

User defined variables would also be possible. While there is no concrete specification for such variables yet, they could be very broad in functionality. They could test for total CPU time used by a working module or even query the amount of data sent across a particular data connection. The only limitation is currently the implementer's need (which we have found to be *unlimited* at times).

Drag-and-drop style GUI

The approach to coordinating ensembles in MARS suggests a clear path for program generation with an object-oriented graphical tool. The ensembles shown in various figures throughout this document are accurate visualizations of multidisciplinary applications. When implementing the first ensembles, before the specification language defining leaders and sub-modules was complete, the module connections had to be defined by hand. If one module was to connect to another module, an explicit message had to be sent from the leader to one of the two modules indicating that a connection to the other was required. For the leader to effect the modules' connection, it had first to identify the actual address of both modules. Then the leader would send the address of one module to the other, followed by a "connect" message for each connection between the two modules in question. To help with this process, *sketches* of ensemble interconnections were made. As the connections indicated by the sketches were coded into the leader, it was indicated on the ensemble sketch.

In the current setup, the process is much simpler. To connect two modules, the leader executes a `leader->link("moduleA.out", "moduleB.in")`. That is it. Once inputs and outputs are defined, connecting them is a snap. The `leader->link()` call generates all the essential lookups, messages, and calls to the runtime layer to connect the two modules.

The initial sketch-and-connect process suggests that a GUI-based tool would be appropriate for designing and implementing MARS ensembles. Simply dragging and dropping modules and their connections into an ensemble would be enough for many types of module collections, including multistage pipelines and cyclical feedback applications. More complex applications, such as those with conditional execution paths, would likely still require significant execution path programming within the leader, but the entire ensemble connection scheme could still be done in the GUI in any case. We envision that such a design tool might look similar to Figure 21.

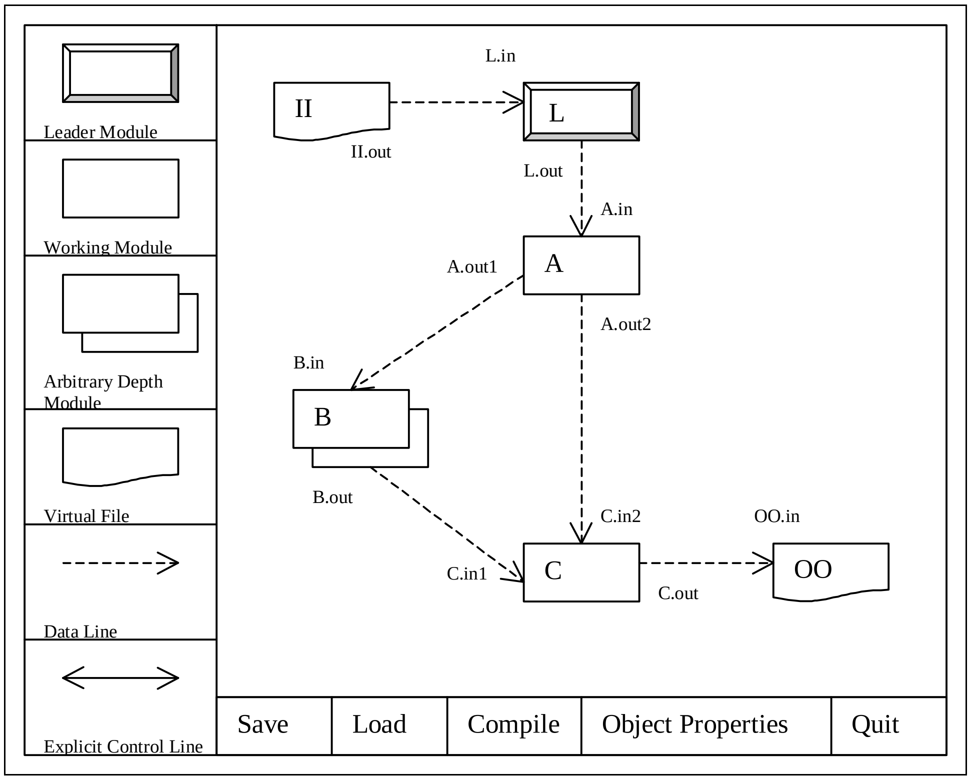

Figure 21. Proposed GUI-based MDO programming tool

VII. Final words

The multidisciplinary optimization problem is not a problem of heterogeneous operating systems, development languages, programming paradigms, or platforms. It is more accurately a problem of drawing together components from *heterogeneous disciplines* to determine answers beyond the scope of any single discipline's contribution. The potential realizable, real-world solutions that can come from multidisciplinary optimizations are truly limitless.

With the advent of teraflop architectures, computational ability has reached beyond the imaginations of programmers even a few decades ago. Computational brute force, however, is not enough. There is an ever growing *crisis of depth* in expert knowledge. Experts are knowledgeable within their domains, but as the accumulation of human knowledge (including raw data) accelerates, experts must further limit their knowledge domains. Perhaps the greatest potential for novel solution discovery lies in composing expert knowledge from multiple disciplines into aggregates capable of solving problems much larger than those encountered by a single expert in the course of his or her career.

Expert knowledge is more and more frequently embodied within computational models. The ability to combine these models is extremely important to solving ever larger problems and to provide innovative answers where we could not even begin to ask appropriate questions before. Take the U.S. Accelerated Strategic Computing Initiative (ASCI). By exploiting knowledge and models from multiple domains, we move closer to complete and accurate simulations for predicting weapons systems performance, safety, and reliability. *Better weapons* are not the answer, but the ability to *substitute simulations* and models for nuclear testing is important to everyone.

There is a common lament that "the weatherman is always wrong". This would not be as accurate if there were better ways to predict. When climate and weather predictions are wrong, the costs can be very high in terms of lives and property and general discord. How can better predictions be made? By simply integrating more data from more sources? Of course, but the predictive power of temporally removed information extends only so far, and increased precision in weather data (state information) can only yield so much benefit[21]. By including multiple diverse models, new perspectives can be gained. The lingering gaps in current practices are more likely to be filled by broadening perspectives, rather than simply adding more information to existing models.

Perspectives are broadened when we leverage useful models from diverse disciplines to solve existing problems. *Experts in their fields know what they know best*--a glaring truism, but important nonetheless. By composing expertise we develop multidisciplinary optimizations. *Optimal* may be in terms of computational resource efficiency, temporal efficiency, accuracy, and solution applicability.

We need experts. Now that experts can compartmentalize much of their expertise, there is an increasing need for composers and composition tools. Hopefully MARS and its discussion will enhance the expanding composition dialogue.

---

[21] And somewhere in China, a butterfly flaps its wings.

References

Adler95	R. Ader, "Distributed Coordination Models for Client/Server Computing," Computer, April 1995.
Bal96	H. Bal and M. Haines, "Approaches for Integrating Task and Data Parallelism," Technical Report IR-415, Vrije Universiteit, Amsterdam, December 1996.
Carriero89	N. Carriero and D. Gelernter, "Linda in Context," Communications of the ACM, April 1989.
Chapman94	B. Chapman, M. Haines, P. Mehrotra, H. Zima and J. Van Rosendale, "Opus: A Coordination Language for Multidisciplinary Applications," ICASE Technical Report 97-30, June 1997.
Haines94	M. Haines, B. Hess, P. Mehrotra, J. Van Rosendale and H. Zima, "Runtime Support For Data Parallel Tasks," *Proceedings of The Fifth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.
Haines95a	M. Haines and P. Mehrotra, "Exploiting Parallelism in Multidisciplinary Applications Using Opus," *Proceedings of the Seventh SIAM Conference*

on Parallel Processing for Scientific Computing, San Francisco, CA, February 1995.

Haines95b	M. Haines, P. Mehrotra and J Van Rosendale, “SmartFiles: An OO Approach to Data File Interoperability,” ICASE Technical Report 95-56, July 1995.
Hunt96	E. Hunt, S. Piper, R. Nemani, C. Keeling, R. Otto and S. Running, “Global Net Carbon Exchange and Intra-Annual Atmospheric CO_2 Concentrations Predicted by an Ecosystem Process Model and Three-Dimensional Atmospheric Transport Model,” Global Biogeochemical Cycles, September 1996.
ICASE95	<u>ICASE Research Quarterly</u>, Vol. 4, No. 1, March 1995.
Kleiman96	S. Kleiman, D. Shah and B. Smaalders, <u>Programming with Threads</u>, SunSoft Press, Mountain View, CA, 1996.
Lottor88	M. Lottor, “TCP Port Service Multiplexer (TCPMUX),” *Request For Comments 1078 (RFC1078)*, November 1988. Available at: <u>http://www.faqs.org/rfcs/rfc1078.html</u>

Mehrotra94	P. Mehorotra and M. Haines, “An Overview Of The Opus Language And Runtime System,” ICASE Technical Report 94-39, May 1994. (Also appears in *Languages and Compilers for Parallel Computers*, Springer-Verlag Lecture Notes in Computer Science, vol 892, Pages 346-360.)
Perrochon97	L. Perrochon, G. Wiederhold and Burback, “A Compiler for Composition: CHAIMS,” *Fifth International Symposium on Assessment of Software Tools and Technologies* (SAST'97), Pittsburgh, PA, June 3-5 1997.
Tornabene98a	C. Tornabene, P. Jain and G. Wiederhold, “Software for Composition: CHAIMS,” *Workshop on Compositional Software Architectures of OMG, DARPA and MCC*, Monterey, CA, January 6-8 1998.
Tornabene98b	C. Tornabene, D. Beringer, P. Jain and G. Wiederhold, “Composition on a Higher Level: CHAIMS,” *unpublished*, Dept. of Computer Science, Stanford University.
Wiederhold92	G. Wiederhold, P. Wegner and S. Ceri, “Towards Megaprogramming,” Communications of the ACM, November 1992.