Pipeline Expansion in Coordinated Applications *

Carl Bartlett, **Neal Sample**, and **Matt Haines**

Department of Computer Science
University of Wyoming
Laramie, WY 82071

May 7, 1999

Keywords: multidisciplinary application; runtime system; legacy code; MARS; pipeline expansion;

Abstract

To support the integration of existing programs into multidisciplinary applications, we have constructed the Multidisciplinary Application Runtime System (MARS). MARS supports legacy code integration, heterogeneous execution environments, conditional execution flows, dynamic module invocation and realignment, and dynamic binding of I/O paths. It also provides a simple specification language to script the overall execution of the multidisciplinary application. In this paper we describe extensions to the MARS system to support dynamic expansion of individual modules in a pipelined multidisciplinary application. Specifically, we focus on the software infrastructure needed to support dynamic module expansion in a pipelined multidisciplinary application.

1 Problem

There is an increasing trend towards building large models by coordinating a set of existing applications rather than creating new ones. This trend permits the creation of coordination models that can solve larger and more complex problems. Coordination models and applications are often used by the scientific community because expert knowledge in one field, as embodied in an application, can be combined with knowledge from another field to solve problems of increasing scope. These types of coordination models are called multidisciplinary optimization problems and the resulting coordinated applications are referred to as multidisciplinary applications (MDAs). Many of the available components used to build an MDA are heterogeneous in terms of source language, data format, and scope of functionality. Thus, coordinating such applications requires a sophisticated software infrastructure.

One example of an MDA problem is aircraft design, which requires expertise from many different areas of expertise, including structural design, thrust, airflow, control, and lift, with each discipline creating its own application. By combining all of these applications, the problem of designing an aircraft becomes simpler because each task is integrated into a single MDA.

In general, the construction of an MDA must deal with three problems: managing concurrency and parallelism, data exchange, and functional control. For traditional methods of building MDAs, managing concurrency usually consists of serializing the component modules to ensure consistency, though at the cost of wasted parallelism and lower efficiency. Data exchange is commonly done via the underlying file system, resulting in slow data communication between the modules. Functional control is responsible for tasks such as module invocation and termination, and is typically accomplished with native script languages or system calls. Overall, the existing methods for building MDAs rely on a set of techniques and tools that are not designed or optimized to make building an MDA easy nor to make the resulting MDA efficient.

The *Multidisciplinary Application Runtime System (MARS)* was designed to provide a software infrastructure for simplifying the design and improving the ef-

*Supported by NSF CAREER Award ASC-9623904

ficiency of multidisciplinary applications. To this extent, it provides for concurrency & parallelism, data exchange, and functional control. It supports abstract data communication on asynchronous partially ordered channels, remote service requests, dynamic module invocation and various other primitives, control and analysis.

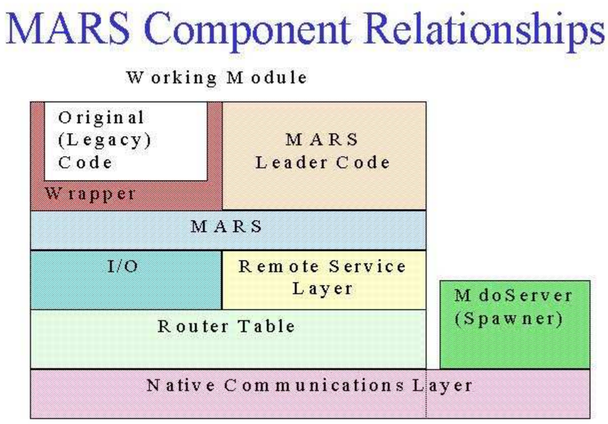
MARS is used as a composition tool, bringing applications from various disciplines together to form an MDA. Each application is transformed into a module and connected to another to form a pipeline through which the data is processed. Pipeline expansion refers to creating multiple instances of a particular module so that bottlenecks can be avoided. The software automatically maintains status of the expanding pipeline and adjusts communication channels accordingly.

This paper describes the changes made to the infrastructure of MARS to allow pipeline expansion and how the actual expansion is works. In Section 2 we give an overview of the components of MARS and in Section 3 describe the changes made to infrastructure and walk through an example of how the pipeline expansion works. In Section 4 we demonstrate the features and benefits of pipeline expansion using a well-known graphics pipeline application. In Section 5 we review related works and close in Section 6.

2 The MARS SYSTEM

MARS is comprised of six different components: the wrapper, the native communication layer, the message routing layer, the remote service layer and remote invocation layer and the MARS module layer. Each of these components is either a preprocessing component, a runtime library or a support layer for remote invocation. The MARS component relationship is shown in Figure 1.

Overall, MARS works by first wrapping the component modules so that I/O and control paths can be redirected into the MARS layer. A **leader** code is then constructed to orchestrate the execution of all wrapped modules and provide them with the information necessary to construct communication channels between themselves. After that, the codes are then initiated and controlled by the leader, and communication is done directly from module to module, without involving the native file system. The following subsections provide a more detailed description of each of these MARS components.



2.1 Wrapper

The wrapper is the preprocessing component of the MARS system and is responsible for transforming the legacy module into a MARS module server. Once the module is wrapped with the runtime routines provided by MARS, it is ready to be instantiated by a leader module.

The wrapper runs as a source-to-source translator, taking a legacy code as input and outputting the same code but with code added to manipulate I/O and control paths. For example, all external entry points are wrapped into remote service request handlers, and a generic dispatch program is installed as the new "main" procedure. This dispatcher then waits for remote invocation instructions from the leader and executes them as they arrive.

In addition to changing the control points, the wrapper re-directs all program I/O to the distributed I/O layer within MARS. This is necessary because most stand-alone applications would normally write to the local file-system, yet we want the data streamed over sockets to another module.

An important characteristic of the wrapper and the MARS system is to enable a wrapped program to run in "stand-alone" mode, in which it should behave exactly as it did before being wrapped. This allows for *single-source* development of these legacy modules, which greatly simplifies code development and maintenance since each change does not have to be posted to multiple versions of the same program.

2.2 Distributed I/O

Most scientific applications use disk-based I/O routines to generate data that is later analyzed or used as input by another program. There are a variety of problems with disk-based I/O, but the two main problems are the inability to share data between modules that do not have access to the same file-system and the performance degradation of using files for passing data between modules.

To combat these problems, MARS integrates distributed input and output routines in an I/O layer called DINO (not to be confused with the DINO language). The DINO layer abstracts files from the physical devices and substitutes existing native communication channels instead. These communication channels are then routed through a dynamic router table that keeps track of which communication channels are connected together. In addition to providing a way to send data once destined for a file through a socket to an arbitrary recipient, DINO also permits multiplexing the data to multiple destinations, including files. So, for example, an original call to `fwrite` might be transformed into a DINO call `dfwrite` that sends the data not only to another module over a TCP/IP socket, but to a named file as well.

2.3 Native Communication Layer

The native communication layer is the foundation for message passing in the MARS system, but is considered independent of MARS since different types of schemes can be used to pass messages. In our current implementation of MARS, the TCP/IP transport layer is used, which is available on all systems today. However, other transport schemes could be used as well, such as UDP/IP or FastMessages.

The choice of TCP/IP as our transport layer allows MARS to be executed over many different network link layers, including 10/100 baseT Ethernet and fiber ATM. Currently, the only requirement of the native communication layer is guaranteed ordered message passing along any single channel, though this restriction could be eased with additional code to ensure this ordering.

2.4 Router Table

The *router table* is our novel data structure used to install and track communication channels, and works much the same way as a network packet router table. The idea is that a sender has an outgoing channel address to which data will be posted. The router table contains a setting that connects this channel to some physical communication descriptor, such as a socket or file descriptor. At any time, and without knowledge or cooperation of the sending client, the router table can change this mapping and effectively change the destination of the output stream. The ability to dynamically alter the output destinations for our virtual I/O channels permits us to implement expanding pipelines, and will be discussed in more depth in Section 3.

In order to keep track of when the router table should alter the destination for a given channel, the router table keeps a *schedule* for each channel. This schedule allows the local router tables to modify themselves at each synchronization point in order to avoid constant communication with the leader. This means that a schedule not only contains information about how to map a given I/O channel to a given physical I/O descriptor, but provides a way of changing this mapping locally so that constant updates from the leader are not necessary.

Another function of the router table is to track files that are associated with a particular channel. At runtime the leader provides a set of filenames and then associates each filename with a particular channel. When the legacy code is executed and a particular file is opened, the new DINO open function searches the router table's filenames. If the filename is found the new file structure is associated with that channel, otherwise the file structure is associated with a file pointer as it would be in the original legacy code.

2.5 Remote Service Layer (RSL)

The remote service layer (RSL) is the primary message passing layer for structured messages. RSL communication is ordered between modules along virtual connections because the native communications layer is similarly ordered. This ordering of RSL messages is required to ensure proper synchronization and execution of the entire MDA. 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.

The RSL also takes charge of processing messages, which is to dispatch the correct service on the remote node. Given the set of language primitives, the RSL translates and implements all requests in all RSL messages. It modifies the router table, invokes native procedures within the wrapped module, and establishes data channels.

The RSL also performs many functions requested by the MARS programmer, such as logging. Log files are opened and written, and the level of logging is set using the RSL.

2.6 Leader Code

The MARS module layer is used to create leader codes. Leader codes are modules that coordinate all other modules in a multidisciplinary application. The MARS leader is a set of language primitives, control functions and a data structure that represents the connected pipeline. The construction of a leader code is broken into three areas. The first is declaration and assignment using the language primitives, the second is setup of instances of the modules, and the third is functional control of the system.

The declaration area uses language primitives to declare modules and define the properties of the modules. It is similar to defining a class in C++ and setting methods and variables associated with the class. The properties of a module type are used when a module is instantiated to create the correct connections.

The next part of the leader code controls the initialization of the data structure used to represent the pipeline and instances in the pipeline. Once an instance of a module in the pipeline is declared and recorded, data channels are created that are associated with a particular module. These data channels are then linked together according to more language primitives. Then the entire pipeline is instantiated and connected together using the RSL layer.

The final area of the leader code is the functional code which describes the coordination of the program by executing stages of the pipeline and directing the data according to the programmer's plans. This is done by passing messages to modules directing them to execute some function, and sending router table updates which redefine the data channels.

3 Pipeline Expansion

Many coordinated applications take the form of a pipeline, where the output of one module is used as the input for the next, and so on. Once a pipeline is established and data begins to flow, it often becomes apparent that there is a bottleneck at some "stage" of the pipeline. This bottleneck has the effect of eliminating the pipeline parallelism that might otherwise be exploited, and reduce the overall execution time of the pipeline to the time it takes to execute that single stage.

While testing an earlier version of MARS [6], we came up against this exact problem, and saw the opportunity for an interesting and useful optimization, namely *pipeline expansion*. The idea behind pipeline expansion is to make duplicate copies of the slow stages in a pipeline, thus eliminating the bottleneck. While it would be possible for a user to continually re-structure their code in an attempt to continually remove these bottlenecks, such an approach suffers from two major problems. First, the user must be able to spot the bottlenecks, which may not always be easy. Second, and more importantly, the location of these bottlenecks *change over time*, making a static approach only somewhat useful.

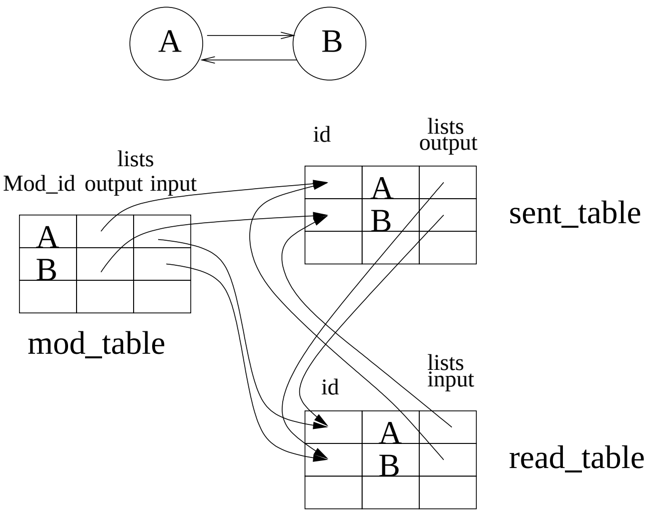
In light of these observations, we set out to modify MARS so that it would be able to recognize pipeline bottlenecks and expand the pipeline to remove them. Moreover, we wanted our software design to enable changing the pipeline expansion dynamically, and thus be able to adapt to changing application modes.

3.1 Changes

To achieve this new functionality we modified both the router table and the module layers' data structures so that pipeline expansion could be expressed. Several data structures were also added to the module layer, including a table of instantiated modules (`mod_table`), and two connection tables, one for output connections (`send_table`) and one for input connection (`read_table`). The `mod_table` tracks all instantiated module and all the module's input and output connection ids. These ids are indexes into the other tables (`send_table` and `read_table`). The connection tables have a connection id, a list of channel ids, and an index to the `mod_table`. The list of channel ids is a list of indexes in the opposing connection table, which representing connections between channels. Thus the `send_table` has a list of indexes to the `read_table` and similarly for the `read_table`'s connection list.

This set of structures allows information about any part of the pipeline to be obtained easily. Information about a module's connections can be obtained by indexing the `send_table` and `read_table` through the `mod_table`'s connection ids. Similarly information about a module is obtained through indexing the connection table entry's `mod_table` id. Combining this scheme, information about the previous or next module is obtained by indexing a `mod_table` entry's connection id, then indexing an id from its connection list and then indexing that id's `mod_table` id. Refer to Figure 2.

At the setup stage of the leader module, a single stage pipeline is created, with one instance of each module. This initializes the data `mod_table`, `send_table`, and the `read_table`. To expand a stage within the pipeline an instance within the pipeline is identified for expansion and a duplicate instance is created. Using the original instance, all connections are created in two connection tables. To make this clear one instance of a connection will be referenced. Assume that the original module has an output connection O, then a new instance in the `send_table` is created, O'. The entry O's connection list is copied to the O' connection list. Next every `read_table` index in the O' list is updated by adding O' to its connection list. This process happens for all of the module's connections, both input and output connections. Finally using all the connection information new router tables are sent to the duplicate module and to any module that is connected to it, in order to establish the actual connection to the duplicate module.

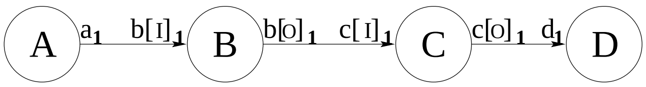


This process ensures that the pipeline connection graph is fully connected, which is necessary for both forward and backward traversals. To ensure the correct ordering of data as it progressed through the graph, a schedule structure was created. This schedule table identifies a particular channel and indicates a policy for either sending or reading from its connection list. These schedules are based on an execution number which is incremented by the leader for every time the functional portion of the code is executed. This allows us to set a range of executions for which a particular schedule is valid. By specifying this range the module no longer needs constant communication with the leader, which is necessary to ensure good performance.

The basic policy used during pipeline expansion is a round robin schedule. This rotates through the connection list sending data across each channel then repeating through the list. However a different policy might be needed if a specific instance with in a stage is bottlenecked. This policy might specify sending and reading data from the faster instance multiple times for every send and read from the bottlenecked instance. By using different scheduling policies, the pipeline can be optimized for a specific run.

3.2 A Walk Through

To further describe the process of expanding a stage, we will follow the steps in expanding the following pipeline where there are declarations of modules A, B, C, and D data channels a_i, $b\,[I]_i$, $b\,[O]_i$, $c\,[I]_i$, $c\,[O]_i$, d_i.



After the initialization and spawning of the modules within the pipeline, Figure 3 shows how the modules are connected. At some time during execution, it is decided that B is a bottleneck and the total execution time would be improved by expanding B's stage. Using the expansion process explained above, B is duplicated with one input and one ouput channel. Then using $b[I]_1$'s connection to a, $b[I]_2$is connected to A via a_2 which is just the process of adding $b[I]_2$ to a's connection list. Figure 4 shows the graph after the addition of $b[I]_2$ to a's list as a_2 and, likewise the process is duplicated for $c[I]$'s connection list to produce $c[I]_1$ and $c[I]_2$.

A, C and the new instance of B are sent new schedules based on this pipeline configuration. The schedule alternate sending data over $a_1$and a_2 and reading data form $c_1$and c_2, based on the execution number. Based on this same execution number each instance of B is executed alternatively.

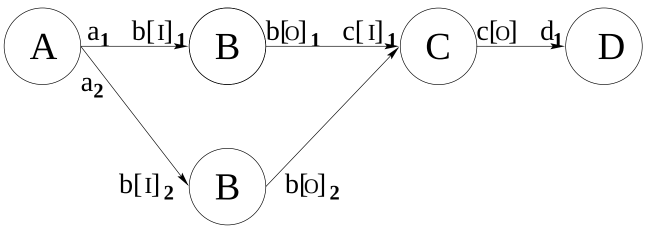


Figure 4

This process of expanding stages can be applied any number of times to any stage. So lets examine the same pipeline but expanded B's stage and C's stage as in Figure 5. Now all four modules are sent new schedules. A's schedule indicates that it will alternate between every three executions. Each schedule sent to B will direct data to C based on odd or even execution numbers. C's schedule is similar to A's in that it alternates by three determining which instance of B is should read from, but C's out put schedule alternates based on odd and even. Finally D's schedule alternates on odd and even exactly like C's output schedule. This scheme ensures that data isn't lost or become unordered during execution.

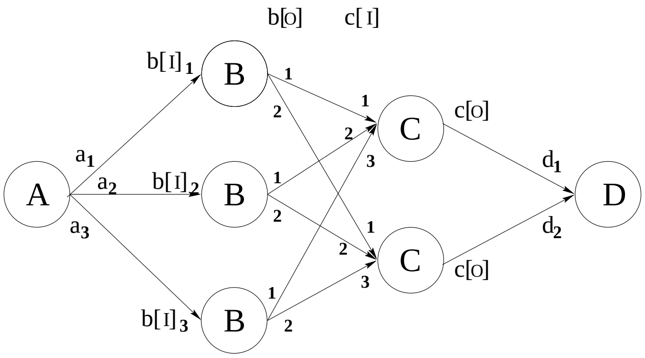


Figure 5

4 Application

To be able to expand a stage of the pipeline, the data must meet one requirement; iterations must be independent of one another. We refer to these independent iterations as "granules," and assume that they can be sent to any instance of a pipeline module that we've expanded.

For our target application, we will examine a famous image transformation problem that reads in multiple pictures and processes them using several techniques arranged in a pipeline. Three different transformation techniques (or pipeline stages) are used: *is*, which increases sharpness of the image; *po*, which posterizes the image; and *rn*, which reduces the noise in the image. We can identify a picture as a granule since it is independent of any other picture, and represents the unit of work that is sent through the pipeline.

To create our MDA, we add wrap these modules to create MARS MDA components and add a fourth component: one that writes images to a file (`wr`). In addition, we construct a leader module to control the overall execution of the image transformations. Figure 6 shows the pipeline's configuration at the start of execution.

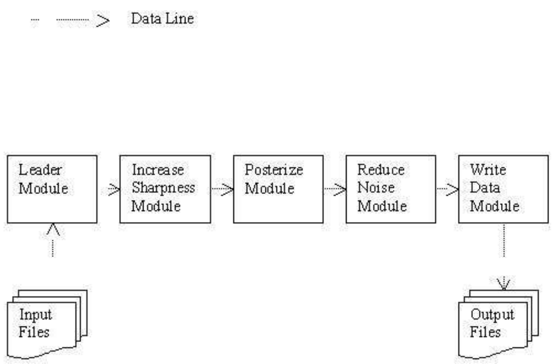
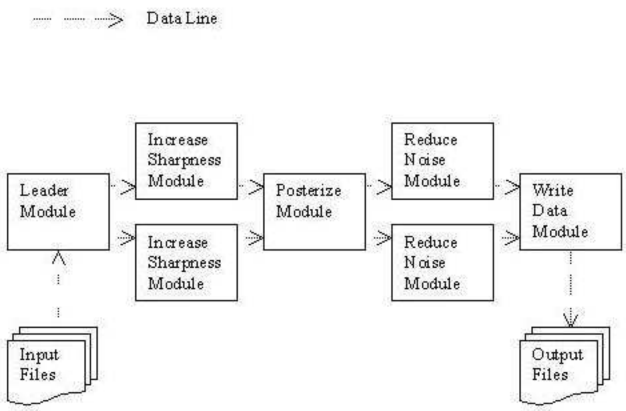


Figure 6

As the pipeline begins execution, we determine that *is* and *rn* are bottlenecks within the pipeline and, as a result, both *is* and *rn* are expanded to speed up the execution time for the entire pipeline. The resulting pipeline is shown in Figure 7. The leader now sends alternating pictures to each instance of *is*, according to a a round-robin schedule. Similar schedules are sent to *po* and *wr*. These schedules indicate which incoming and outgoing channels the module will read/write the data on. The schedule instructs the leader to pass data to is_1, and based an a similar schedule passed to *po*, *po* will read data from is_1. By scheduling the entire pipeline's route, the data will arrive at *wr* in the proper order. It may occur that *wr's* channel from rn_2 has data on it, but as long as *wr's* schedule indicates that the next granule is expected from rn_1, *wr* waits for data on rn_1. This ensures the correct ordering of data.



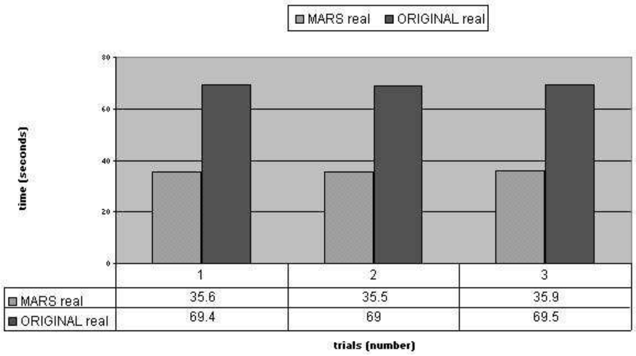


Chart 1

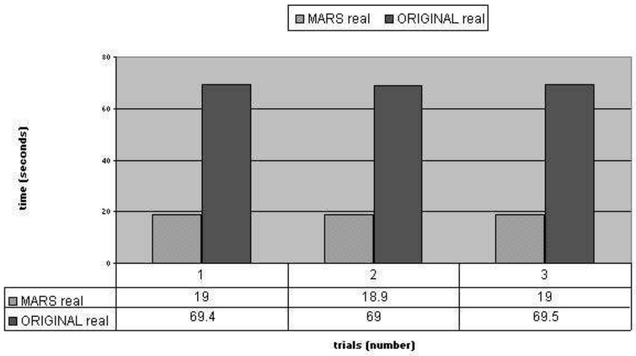


Chart 2

As our data [6] in Charts 1&2 indicate, the expansion of a pipeline can decrease execution times depending on the execution times of each module. Chart 1 shows that on a Sun test bed, the MARS modules executed almost twice as fast as the original setup. However, with a three stage pipeline, we were hoping for more pipeline parallelism and something closer to a threefold improvement in execution time (hey, we're greedy). But it was found that *is* and *rn* take a significantly larger to execute than *po* and *wr*. Our obtained performance of a twofold improvement now makes sense: two stages of the pipeline are dominating the execution time.

By expanding the pipeline, *is* and *rn* were expanded to a depth of three and the execution times for the entire pipeline decreased, making the pipeline over $3\frac{1}{2}$times faster. The speedup in the pipeline is now limited by the speed at which both the leader and the *wr* module could read and write to the disk (i.e., I/O bound). The system could execute in constant time on any number of pictures if a simple reader module where created. This theoretical result assumes that the system resources could handle large number of instances of each module.

Extending MARS with the capability of dynamically expanding a stage at runtime instead of at implementation allows the same improvements in execution times and also gives the programmer the ability to steer the execution by manipulating the pipeline as he or she sees fit. Dynamic expansion also allows for scheduling and expansion based on the decisions made by some outside application that monitors the system resources to best fit the pipeline to the execution times of each instance. By using such a monitoring application, single execution speedup is accomplished. Refining the pipeline no longer takes profiling and recompilation. The MARS leader can refine the connection graph during execution.

5 Related Work

MARS presents an approach for providing a software infrastructure that supports execution of multidisciplinary applicatins. In doing so, it must address multiple issues of coming task and data parallelism. There are a number of other projects that are looking to provide useful models for integrating task and data parallelsim, including Fx [7], Orca [8], Opus [2,3,9].

Fx provides Fortran language extensions that allow data parallel sections of code to communicate through channels (similar to MARS). Each of these data parallel executions is called a task, and communication between tasks (via channels) occurs only at task entry and exit points, resembling the way in which data is communicated between a calling program and a subroutine. Although Fx is good for the design of new task parallel codes, it is not particularly well-suited for MDA applications [1]. 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 MDA builder is once again responsible for handling communication. In effect, a static (where modules are never added or removed) MDA 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).

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 [1]. 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.

The Opus language specification was designed for specifying how to create and ex ecute multidisciplinary applications at a language level [1]. Opus takes a data-centric approach to the MDO problem, introducing the ShareD Abstraction (SDA) for coordination and communication. SDAs act as data repositories, communication channels, and occasionally as compute nodes within Opus ensembles. While Opus is geared at MDA execution, it differs significantly from MARS in that is is a language specification, whereas MARS provides a software infrastructure for MDA execution. Thus MARS may even be used as the compilation target for Opus.

Another runtime approach to providing MDA support can be found in the CHAIMS project [4,5]. While similar in goals to MARS, CHAIMS operates at a much higher level of abstraction. Rather than providing a low-level infrastructure to support legacy module interactions, CHAIMS takes the Corba route of providing objects that know how to perform some task and tell others who might inquire about it. The idea is that there would be a large collection of these CHAIMS objects just sitting "out" in the computation space, and an MDA designer need only query them for the correct functionality and then call upon them for work. This approach is not performance-oriented (as we've seen from Corba), and suffers from the problems of not readily accepting legacy modules.

6 Summary

The primary goal of the MARS project is to build a software infrastructure to support multidisciplinary applications. These applications are formed from collections of modules that often include not only newly-created modules, but also legacy programs. The working modules that form an ensemble cooperate in a heterogeneous execution environment, frequently are composed of clusters of workstations.

By extending MARS to include dynamic expansion, complicated leader codes are eliminated, and replaced by simple function calls that perform the stage expansion operations. Dynamic stage expansion also allows outside applications that monitor the system resources to direct the expansion. This allows single execution speedup instead of profiling and optimizing the source code.

References

[1] H. Bal and M. Haines, "Approaches for Integrating Task and Data Parallelism," Technical Report IR 415, Vrije Universiteit, Amsterdam, December 1996.

[2] 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.

[3] 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.)

[4] C. Tornabene, D. Beringer, P. Jain and G. Wiederhold, "Composistion on a Higher Level: CHAIMS," *unpublished*, Dept. of Computer Science, Stanfor University.

[5] G. Wiederhold, P. Wegner and S. Ceri, "Towards Megaprogramming,"Communications of the ACM, November 1992.

[6] N. Sample, C. Bartlett, M. Haines, "MARS: Runtime Support for Coordintated Applications," Proceedings of the ACM Symposium on Applied Computing, San Antonio, TX, Feb 28- Mar 2, 1999.

[7] T. Gross, D.R. O'Hallaron, and J. Subhlok, "Task Parallelism in a High Performance Fortran Framework," IEEE Parallel and Distributed Technoloty, pg 16-26, Fall 1994.

[8] S. Ben Hassen and H.E. Bal., "Integrating Task and Data Parallelism Using Shared Objects," ACM International Conference on Supercomputing, Philidelphia, PA, May 1996.

[9] B. Chapman, M. Haines. P. Mehrotra, J.V. Rosendale, H. Zima, "Opus: A coordination language for multidisciplinary applicatins." To appear in Scientific Programming, 1996.