Language Support for Multidisciplinary Applications*

Piyush Mehrotra[a] *John Van Rosendale*[a] *Hans Zima*[b]

[a]ICASE, MS 403, NASA Langley Research Center, Hampton VA. 23681 USA
E-Mail: {pm,jvr}@icase.edu

[b]Institute for Software Technology and Parallel Systems,
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria
E-Mail: zima@par.univie.ac.at

Abstract

With the growing availability of heterogeneous networks of computing systems, multidisciplinary applications (MDAs) are becoming increasingly tractable. Such applications combine a number of programs representing different "disciplines" into a coherent, integrated system of interacting processes. In this paper we study the characteristics of MDAs and derive a corresponding set of language requirements, focusing on modular system design and multi-level parallelism. In addition to providing a brief overview of a number of languages addressing various aspects of the problem, we present a detailed overview of the Opus language, which was specifically designed with the goal of supporting the requirements of MDAs.

1 Introduction

With the advent of teraflop computing, complex simulations in many disciplines are becoming tractable, and there is growing interest in "multidisciplinary" applications (MDAs). These are applications spanning traditional disciplines and involving both the "science" occurring in each discipline separately, and the potentially complex interactions between disciplines. One area of interest is multidisciplinary simulations, such as coupled ocean-atmosphere models. Another area is multidisciplinary optimization, in which an engineering design is evaluated and optimized simultaneously in several engineering disciplines. For example, one might simultaneously optimize the structural and aerodynamic properties of an aircraft.

In addition to the intrinsic scientific challenges arising in multidisciplinary applications, there is a computer science or software challenge. Programs for multidisciplinary applications are typically created by pasting together legacy software from the constituent disciplines, though new software is also written as needed. This approach raises complex software design and engineering challenges, and there are issues involved in extracting and exploiting the multiple levels of parallelism available as well.

In this paper we consider the design of languages supporting various aspects of multidisciplinary applications. Our target environment is a heterogeneous distributed network of execution resources, consisting of machines ranging in scale from workstations to large scale multiprocessor architectures. The software challenge in MDA-style applications is that of coupling a set of large programs into a coherent whole, and coordinating their execution. What is needed is, in effect, a specialized kind of coordination language. The paper describes the characteristics and needs of MDA-style applications, and the language features needed to support them. It also describes, in some detail, the authors' Opus language, one potential response to the needs of this application area. We focus on the linguistic approaches for expressing MDAs; software

*The work described in this paper was partially supported by the ESPRIT IV Long Term Research Project 21033 "HPF+" of the European Commission, by the Austrian Ministry for Science and Transportation under contract GZ 613.580/2-IV/9/95, and by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 and NAS1-97046, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

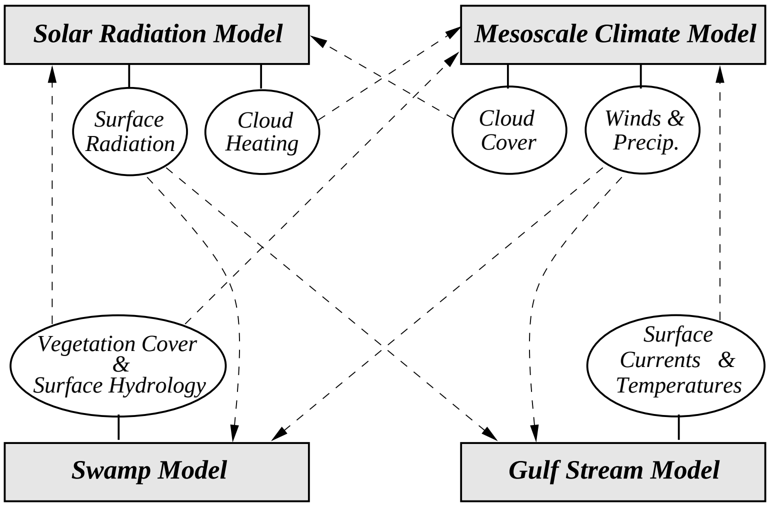

Figure 1: Models for environmental simulation of the Florida swamp ecosystem.

frameworks which use scripts and other mechanisms for gluing programs together are outside the scope of this paper.

The paper is structured as follows. In Section 2, we discuss the salient characteristics of MDAs and the resulting language requirements. Section 3 presents a brief overview of a number of languages and programming paradigms that address these issues. In Section 4 we give a detailed overview of Opus, an object-based language that supports modularity and integration of task and data parallelism. Section 5 concludes the paper.

2 Characteristics of Multidisciplinary Applications

In this section we look at the characteristic features of multidisciplinary applications along with the kinds of facilities any language or environment suited to this application area would need to provide. As a representative example, consider environmental simulation of southern Florida. Figure 1 shows four distinct models (shown as rectangles) representing different aspects of the Florida ecosystem. The goal is then to interconnect these four models into a combined "multidisciplinary" model for the ecosystem.

Each of the four models shown is fairly complex, and is composed of various submodels. We represent the submodels relevant here as ovals attached by a solid line to their parent model. For example, the mesoscale climate model contains a submodel for wind and precipitation and another for cloud cover. The wind and precipitation submodel affects the swamp model directly, while the cloud submodel affects the solar radiation model, and hence indirectly affects the swamp model.

Correctly simulating these complex couplings, shown as dashed lines, is the essence of multidisciplinary applications. The combined Florida environmental model will subsume all of the original models together with their various couplings, and will necessarily involve a complex mixture of legacy and new software. Other examples of MDAs include global climate modeling and design optimization of aircraft. The latter is discussed in greater detail in Section 4. First, however, we look at some of the characteristics of multidisciplinary applications and describe the language support required for such applications.

Modularity

Multidisciplinary applications are typically written by combining a number of programs, each modeling a single discipline, into a larger meta-program. A team of programmers is often involved, comprised of experts from the various disciplines. Each team then works on the program corresponding to its discipline, but must also have some cognizance of the model being developed or modified for the other disciplines involved.

The natural way to support this is with a module facility, as available in many languages, particularly object-oriented languages. In some cases modules may encapsulate legacy code, while in others, this will be newly written software. For both cases, the linguistic support needed includes:

- independent name spaces, so that the types, definitions and variables in different modules do not collide,
- read/write access protection for shared data,
- clear interfaces, to facilitate mixing and matching of different modules for different simulations, and
- support for multiple source languages.

These mechanisms allow the individual disciplines codes to be developed independently once the interfaces have been specified. Such plug compatibility of the codes permits the discipline experts to vary the fidelity of the computation based on the phase of the simulation. In other words, low fidelity codes can be utilized when only coarse solutions are required while using high fidelity codes when a finer degree is necessary.

Parallelism

Multidisciplinary applications exhibit parallelism within and across the discipline codes, and it is important to be able to exploit both kinds of parallelism. The reason is not so much to exploit every bit of available parallelism but instead to improve the overall processor utilization and minimize sequential bottlenecks.

In general, the computational demands of the individual codes vary widely. Some discipline codes are computationally intensive and also exhibit a great deal of internal parallelism. Others may be sequential and much less computationally demanding. In the Florida ecosystem model described above, the mesoscale climate model contains loops over thousands of mesh points, and is both computationally intensive and highly parallel. The other discipline codes involved may use much simpler models, and may be inherently sequential.

It is natural to view the outer level of a multidisciplinary application as a set of processes executing independently in a heterogeneous environment. Assuming the language provides the needed support, there is real gain in this approach. Suppose, for example, that the mesoscale climate model in the above example runs on a large scale multiprocessor. If the other discipline codes can be executed concurrently on other processors, the computationally intensive mesoscale climate simulation will not have to be interrupted while awaiting results from the other three discipline codes. Also, communication need occur only once per time step, rather than three separate times. Minimizing the number of communications phases is particularly important in geographically distributed simulations.

To support this kind of coarse-grained task parallelism, the language needs only simple mechanisms. These include barrier synchronization, waiting for the termination of a process or other events, and testing the status of some activity. Typical coordination patterns employed in such systems are the master/slave and producer/consumer relationships.

Another issue is support for dynamic process creation. If the overall structure of the application can be statically determined, then creating all the required processes at the start of the application may suffice. However, if the analysis or the resource requirements are data dependent and can only be determined at runtime then dynamic process creation will allow the application to create the appropriate processes as and when needed.

At the same time, it is critically important to support fine-grained parallelism within each discipline code. One natural approach is use of a data parallel language like HPF or HPC++. Alternatively, one could express the fine-grained parallelism using MPI or related notations for describing fine-grained interprocessor communication.

Data Sharing

Since multidisciplinary applications can be large numerical computations, communication and data sharing are critical. There are two kinds of data to be communicated:

- process status
- arrays of numerical values and associated data structures

These two kinds of interactions are quite different, not only in their nature but also in the amount of data that needs to be communicated. Thus, they may best be handled by different language mechanisms such as shared data structures or message passing.

Another issue that frequently arises in such applications is the translation of data values and formats between modules corresponding to different disciplines. In the Florida ecological model, for example, the mesoscale climate model may store pressure, humidity and temperature on coarse latitude-longitude grids, while the swamp biology model may use a completely different grid, and perhaps different units. Thus transferring values between models may entail complex interpolation and conversions of values. The best way to implement this is a sequence of filters to perform the various data translations, rather than surgically altering the discipline codes.

Summary of Language Requirements

As we have argued, the most critical language requirements for multidisciplinary applications include support for modules, parallelism and inter-module communication and synchronization. In more detail, the following features are needed:

- **Module Facilities:**
  - Separate name spaces
  - Specified interfaces
  - Multiple language bindings
- **Process Facilities:**
  - Multiple, independently executing processes
  - Ability to spawn, kill, and synchronize processes
  - Support for data parallelism within each process
- **Data Sharing Facilities:**
  - Support for fine-grained signaling between processes/modules
  - Support for coarse grained data sharing or communication

3 Languages and Paradigms: A Brief Overview

In this section, we study a number of languages and paradigms and evaluate their suitability for MDAs. The discussion is structured into three parts. First, in Section 3.1, we consider programming models that support a single address space. Shared memory models, PCF, Linda and HPF belong in this class. The second part (Section 3.2) deals with two languages – MPI and Fortran M – whose execution model provides explicit message passing across disjoint address spaces. The final class of models, which we believe the most promising, are of a hybrid nature: they provide object-based shared data structures for the communication among processes executing in different address spaces. Section 3.3 discusses three languages – CC++, Agora, and Orca – that fit into this category. A more in-depth discussion of this approach will be conducted in Section 4, where we give a detailed overview of the authors' own language Opus.

One should reiterate that in this paper we are focusing on the linguistic support required for MDAs. Hence, we do not discuss the runtime support, e.g., for providing a virtual shared memory, required to implement the models described below on our target execution environment: a heterogeneous distributed network of computing resources ranging in scale from workstations to large scale multiprocessor architectures.

3.1 Programming Models Supporting a Single Address Space

The first class of models provide a single address space, i.e., all processes share the same address space. The focus of these models is then to provide facilities which control and coordinate the access to the shared data. The conceptual simplicity of the single address space model and its proximity to the conventional sequential programming paradigm makes it highly attractive for parallel architectures. We discuss here three approaches, which mainly differ in the way the address space is structured and accessed.

3.1.1 PCF

PCF [2] is a language independent semantic model for parallel extensions of standard imperative programming languages such as Fortran and C. It is based upon the shared memory version of the Single Program Multiple Data (SPMD) program execution model. A program execution begins with a single process until a *parallel construct* is encountered, which allows the execution of a block of code by a *team* of processes. The parallel constructs include *parallel sections* and *parallel loops*. While the members of a team communicate via the global address space, each team member may have private objects inaccessible to other processes. A *worksharing construct* can be used to distribute work among the members of a team.

Apart from the implicit synchronization associated with the built-in parallel and worksharing constructs, PCF provides explicit synchronization mechanisms that operate on shared data. They include locks, semaphores, and events. An evolution of this approach, OpenMP, has recently been proposed by a consortium of hardware and software vendors, as the portable standard API for shared memory programming [12].

Considering the MDA requirements we see that PCF supports the creation of processes in a structured (SPMD) way, and provides adequate coordination facilities based on a shared memory model. However, the demands posed by modularity, heterogeneity, dynamic process generation and the integration of task with data parallelism are not adequately met in this approach.

3.1.2 Linda

The address space of a Linda [1] program execution, called a *tuple space*, is a global, associatively addressed memory. This tuplespace can be accessed via atomic operations for adding, removing and reading of tuples, where the remove and read operations perform a non-deterministic selection based on pattern matching. A special *eval* operation, when applied to a tuple of expressions, generates a set of parallel processes, one for each field of the tuple. These processes evaluate the associated expressions in parallel, generating more data tuples in turn. The tuple space provides an uncoupled communication paradigm, covering communication in space as well as in time: a tuple remains in tuple space as long as it is not removed, independent of the lifetime of the process that created it.

Evaluating Linda from the viewpoint of MDA requirements, it is clear that the language provides basic facilities for process creation, coordination, and communication, including some support for persistence. However, there is a lack of support for:

- modularity and, as a consequence, scalability (the tuple space is flat),
- data parallelism (process creation is unstructured),
- sophisticated data transfers (producer and consumer of a tuple are disconnected), and
- standard data structures such as vectors and arrays.

Some more recent work addresses a few of these issues. In particular, the Linda Program Builder (LPB) is a programming environment that supports parallel program construction and parallel data structures at a high level.

3.1.3 High Performance Fortran

High Performance Fortran (HPF) [10] is a set of Fortran extensions primarily oriented towards the specification of data parallel algorithms. It provides annotations for the distribution and alignment of data, and for the specification of parallel loops.

The discussion of the HPF programming model has to take into account two levels. At the higher level, the execution of an HPF program is modeled by a single process. Even though users directly provide the data mapping through directives and explicitly express data parallelism via array statements, forall constructs and the *independent* directive, the model stresses the important point that HPF programs operate in one global data space.

The second, lower level model reflects an execution model that is part of the HPF language definition and follows the SPMD paradigm for distributed memory machines. It is based upon a set of abstract processors with disjoint address spaces, each of which executes exactly one process. Data distribution directives partition the (global) address space of the first layer across the abstract processors of the second layer; the *on* directive performs the analogous job for parallel execution control and work sharing.

HPF fully supports those aspects of MDAs that are related to the execution of data parallel programs. In addition, it also offers simple tasking features, which allow the specification of parallel sections executing on disjoint sets of processors, without a mechanism for explicit synchronization or communication. Given the rudimentary nature of these facilities, they are not suitable for expressing the complex interactions among asynchronous tasks as required by MDAs. Similarly, even though the module construct of Fortran provides a limited modularization and encapsulation facility, it does not sufficiently meet the demands of MDAs.

3.1.4 Other Mechanisms

Since Dijkstra's work on semaphores and PL/1's event mechanism, developed in the 1960s, dozens of approaches addressing data sharing in a single address space model have been proposed. Much of this earlier work is described in [3].

3.2 Message Passing Across Multiple Address Spaces

Here we consider programming models based on the concept that each process is executing in an address space of its own with no direct sharing of data structures. Interprocess communication is then achieved via special message passing primitives. Below we discuss two approaches: the de-facto standard *Message Passing Interface (MPI)* and Fortran M. The basic features of PVM, Parmacs, and other message passing models are similar to MPI.

3.2.1 MPI

MPI [11] assumes that a fixed set of processes is (implicitly) created at program initialization, each executing exactly one process. There is no explicit mechanism for process creation.

MPI provides 129 functions, most with a number of variants, for interprocess communication. In addition to point-to-point communication, these include collective communication facilities covering barriers, broadcast, gather, scatter, and reduction operations. The scope of a collective operation can be restricted using *communicators*, which define a group of processes and a communication context. The MPI Forum has recently released MPI 2.0 which supports process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, and additional language bindings.

MPI provides some support for MDAs. In particular, the modularity of the MPI programming model can be exploited to map individual discipline codes to separate modules, with only limited recoding necessary in many cases. On the other hand, explicit message passing represents a very low level for expressing interprocess communication. *Message plumbing* is hardwired into codes, requiring a significant recoding effort if communication patterns change. Furthermore, there is no explicit support for data parallelism.

3.2.2 Fortran M

Fortran M [9] extends Fortran 77 with a set of features that support message passing, according to a strictly enforced discipline. Fortran M processes – program modules encapsulating data and code that are executed concurrently – can be combined via *channels*; each channel establishes a one-to-one connection between typed *ports*, essentially representing a message queue[1]. Communication is performed by sending and receiving from ports. Fortran M provides a PARBEGIN/PAREND like construct and a parallel loop for the structured creation of processes; it has HPF-like constructs for the distribution of work and data. By imposing a FIFO discipline on message queues and guaranteeing a sequential semantics for output arguments, determinism is enforced.

In comparison with MPI, Fortran M provides better support for modularity via its process construct; furthermore, structured dynamic process creation is possible, although with a strong emphasis on achieving determinism. Determinism, however, is not a typical feature of MDAs, and many useful coordination structures cannot be easily expressed in Fortran M given the strictness of its approach. Such examples include producer-consumer problems with multiple producers and consumers accessing a bounded buffer, or the variants of the readers-writers problem.

3.3 Shared Data Structures Across Multiple Address Spaces

In this section we discuss a class of object-based models which are hybrid in the sense that they bridge different address spaces by providing shared data structures. More precisely, these models can be described as follows.

Assume a set of processes p_i, operating in disjoint address spaces. Our approach to establishing communication between the p_i is to identify a set of data, D, and a set of *methods*, M, for dealing with the data, and make both D and M accessible to all the p_i. This is similar in philosophy to Hoare's monitors for synchronization on a shared memory machine in that we isolate the variables and methods needed for communication. Of course, the difference is that we are not executing in a shared memory environment, which means that the shared data structure has to be implemented in the framework of a distributed memory or heterogeneous architecture.

Below we will provide a short overview of three languages – CC++, Agora, and Orca; Section 4 will deal in some detail with Opus, another language which falls in this class. The individual approaches can be characterized by their handling of address spaces, the facilities for process creation, the available mechanisms for specifying methods, and the associated execution and synchronization models.

3.3.1 CC++

Compositional C++ (CC++) [6] is a set of extensions for the C++ language. CC++ provides *processor objects*, which are instances of specially designated "global" classes. Processor objects are associated with a unique address space and can be explicitly mapped to physical processors; thus they represent a unit of locality. One or more processes (called *threads* in CC++) can execute on a processor object; they have access to all data in its address space.

The mechanism for sharing data is the *global pointer*, which can be used to point to objects on a different logical processor. Any process that has access to a global pointer can access all the public methods associated with the pointed-to object.

Processes can be created in a structured (PARBEGIN/PAREND, parallel loops) or in an unstructured way; they can be made to execute on the same processor object as their parent or on a remote processor object using a remote procedure call accessing a global pointer. Remote procedure calls are executed in synchrony with the calling process.

CC++ allows multiple processes to execute in parallel inside a processor object. As a consequence, synchronization must be programmed explicitly where needed. The language provides a single-assignment *sync* variable: the attempt to read such a variable results in blocking if it is undefined. Furthermore, the attribute *atomic* for a method makes a call to that method mutually exclusive with any other method call

[1] In addition, many-to-one and many-to-many communication can be implemented using a special language construct.

for the same object.

CC++ features such as processor objects, global pointers, and the synchronization facilities are highly useful for MDAs. However, the lack of asynchronous remote procedure calls and missing support for data parallelism and its integration with task parallelism reduce the applicability of the language for this class of applications. Recently, a consortium of researchers has proposed HPC++ as an attempt to overlay data parallelism and data distribution facilities on top of the explicit tasking facilities of CC++. However, the definition is still not mature and no complete implementation is currently available.

3.3.2 Agora

Agora [5], developed around 1986, allows the specification of *shared data types (SDTs)* in a Lisp-like syntax. The description of an SDT consists of the specification of (1) its data structures, (2) the associated set of methods, and (3) an addressing mode. Agora provides a set of built-in methods (create, destroy, read, write, atomic execute), which can be combined into more complex user-defined methods. SDT objects can be addressed either by using indexing in a linear array scheme, or through the hashing of strings in a hash table.

The built-in methods provide automatic mutual exclusion for access to an SDT object, and the special *atomic execute* method can be used by the programmer to achieve coarser-grain mutual exclusion. Except for these constraints, processes may access the shared data of an object in parallel. Process control is performed using an event-driven scheme for access to SDT objects. Each process is associated with a queue of activation requests which can be activated by other processes.

The Agora implementation uses a single-assignment discipline for variables, thus simplifying the implementation of global data sharing, but at the high price of having to copy when updating data structures.

Agora is one of the earliest systems which uses concepts related to the requirements of MDAs. While some of its features – such as the basic SDT concept and its associated built-in methods – play an important role in later systems, the method of access to SDT objects, the process control discipline, and the single-assignment implementation strategy do not fit well in an MDA context.

3.3.3 Orca

Orca [4] provides an object model very similar to the model used by the Opus language, which will be discussed in detail in the following section. Orca's *abstract data types* provide encapsulated shared data, mutually exclusive access, and conditional synchronization, all of which are also present in Opus. However, Orca is a distributed programming language which – in contrast to Opus – has not been designed with the explicit objective of supporting the requirements of MDAs. Thus, for example, Orca does not provide any support for data parallelism within the objects.

4 Opus

We have recently designed Opus, a language which provides a hybrid programming model for expressing complex multidisciplinary applications. The heart of Opus is a new mechanism, called Share**D** **A**bstraction (**SDA**). SDAs borrow from object-oriented systems in that they encapsulate data and the methods that act on the data, and from monitors in shared memory languages in that an active method has exclusive access to the data of an SDA.

Tasks, i.e., asynchronously executing autonomous activities, are instantiated in Opus by creating instances of SDAs and invoking the associated methods asynchronously. Different SDAs represent distinct address spaces, hence Opus tasks do not directly share data. However, SDAs themselves can also be used as data repositories to be shared among the tasks. A pool of common data may be shared by a set of tasks by creating an SDA containing the data and making this SDA available to all tasks in the set. Using SDAs and their associated synchronization facilities also allows the formulation of a range of coordination strategies for these tasks. This set of concepts forms a powerful tool which can be used for the hierarchical structuring of a complex body of code and a concise formulation of the associated coordination and control mechanisms.

Below we describe the features of Opus and then show how to use these features for a simple application code. A full description of the language features and their usage can be found in [8].

4.1 Features of Opus

Opus introduces a small set of features for defining and using SDA objects and accessing SDA data. It provides language constructs to define SDA types, declare SDA variables, create, initialize, terminate, and save SDA objects, as well as activate SDA methods both synchronously and asynchronously. The syntax borrows heavily from Fortran.

An SDA type in Opus specifies an object structure, containing data along with the methods (procedures) which manipulate this data. An SDA object (which we usually simply refer to as an SDA) is generated by creating an instance of an SDA type. The creation of an SDA involves allocation of resources on which the SDA will execute, the allocation of data structures in memory and any initializations that are necessary to establish a well-defined initial state. The lifetime of an SDA is the time interval between its creation and its termination. During this interval, the SDA exists and can be accessed via method calls. SDA variables are handles through which SDAs are accessed from within a program.

There are two ways of invoking a method of an SDA: synchronously, where the caller is blocked until control returns, or asynchronously, by a non-blocking call. An asynchronous method execution may be associated with an event, which can be used for status inquiries and synchronization. No two method executions belonging to the same SDA can execute in parallel; as a consequence each method has exclusive access to the data of its SDA. A method may have an associated condition clause, specifying a logical expression which guards the method's activations.

An SDA can be saved by copying it to external storage, thus generating an external SDA, which is identified by a unique external name. External SDAs are persistent, having an a priori unlimited lifetime. Saving an SDA thus makes it accessible for later reuse, by loading an external SDA into memory.

We use HPF syntax to support data parallelism within an SDA. Thus, each SDA type definition may contain an optional processors statement, as defined for HPF procedures [10]. This allows the internal data structures of the SDA to be distributed across the processors using the distribution and alignment directives of HPF. The dummy arguments of the SDA methods can also be distributed using the rules applicable to procedure arguments in HPF.

Each SDA instance is associated with a unique (SDA) task, which is the locus of all control activity related to the SDA. The SDA task operates on the resources allocated to the SDA, provides an address space for the SDA's data, and manages the execution of calls to the SDA's methods. The execution of an Opus program can be thought of as a system of SDA tasks in which a task executes a method of its SDA in response to a request from another SDA.

4.2 An Opus Example

In this section we use Opus to encode a multidisciplinary application such as that used for aircraft design. The overall goal of this application is to optimize the design of an aircraft relative to some goal or "objective function", such as minimization of gross weight. This is done subject to constraints such as specified range and payload. The design cycle starts with these constraints and goals, a base geometry and initial values for a set of design variables, such as sweep angle of the wing and thrust of the engines. Then, in each cycle, an analysis phase analyzes the current configuration of the aircraft, as specified by the design variables, to produce a set of output variables, such as lift and drag. The optimizer then evaluates the objective function for this configuration, to produce new values of the design variables. Effective optimizers are Newton-like methods which require "sensitivity derivatives", the derivatives of the output variables with respect to the design variables. This optimization cycle continues until the process converges to a final "optimized" configuration of the aircraft.

We now describe a version of the above application using Opus in which the codes in the analysis phase execute in parallel. The analysis phase has been simplified to the simultaneous optimization of the aerodynamic and structural design of an aircraft configuration. Though a realistic multidisciplinary optimization of

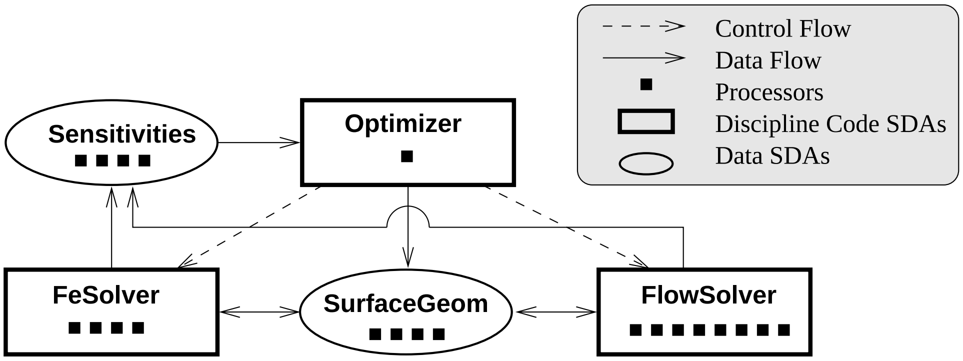

Figure 2: Data flow in a simple multidisciplinary application for aircraft design

a full aircraft configuration would require a number of other discipline codes, including controls, performance analysis, and propulsion, we present this version for the sake of brevity.

The structure of the program, as expressed in Opus, is shown in Figure 2, where the SDAs encapsulating computational activities are represented by rectangles and the SDAs encapsulating data repositories are represented by ovals. We first describe the SDA types which represent the tasks and the data shared by these tasks and then present the main task, the *Optimizer*, which creates all the other SDAs and coordinates the execution of the entire aircraft design application.

4.2.1 The SDA Types

In developing the aircraft design application, the first step is to define the SDA types which encapsulate the discipline codes and act as computation servers and also those that encapsulate the data shared by the discipline codes. As shown in Figure 2, in this application we have two data repositories: the *SurfaceGeom* for sharing geometry and flow data between the two computational tasks, and the *Sensitivities* for storing the sensitivity derivatives. Figure 3 shows the SDA type definition *SGeomSDA* used for the first set of data. It consists of the internal data structures such as *base* , the initial base geometry configuration of the aircraft, *deflected*, the deflected geometry, the current flow solution and other associated items. The procedures defined within the SDA type, e.g., *PutBase*, *PutDeflected*, *GetDeflected*, etc., allow other discipline tasks and the optimizer to access the values of the internal data structures.

As stated in the last subsection, each of the methods in an SDA type may have an associated *condition clause*. This is specified using the keyword **WHEN** and a side-effect free logical expression, as is the case with the *PutDeflected* method in *SGeomSDA*. This clause enables the conditional activation of an SDA method. That is, the condition is evaluated when the method is invoked, and the method can only be activated if the result is *true*. If it is *false*, the method activation request is enqueued until the condition evaluates to *true*. This mechanism allows the SDAs to synchronize access to internal data structures. Thus, in the example here, if the aerodynamic solver invokes the procedure *GetDeflected* to acquire the deflected geometry produced by the current iteration, the call will be enqueued until the structural analysis code has invoked the procedure *PutDeflected* to deposit the current geometry and set the logical variable *DeflectFull* to *true*.

One of the critical features of SDAs is the atomicity of method executions. In order to avoid incoherent states of the data associated with any given SDA, methods are executed as atomic operations. That is, any executing method has complete and sole access to all the internal data structures of the SDA. Thus, the *get* and *put* methods above can access and modify shared variables without interference from other activations of the methods.

Note that the internal parallelism of the data SDAs is specified using the well-developed HPF facilities for data parallelism. Thus, the data structures may be distributed across a set of processors, designated by *P* in the *SGeomSDA*. It is the compiler's job to analyze the code and produce the explicitly parallel, SPMD code for each of the associated procedures.

The SDA types encapsulating the discipline codes can also be specified in a similar manner. For example,

```
      SDA TYPE SGeomSDA(...)
!HPF$ PROCESSORS P(number_of_processors())
         TYPE (surface) base, deflected
         TYPE (flow) FlowSoln
         TYPE (fe) FeModel
!HPF$ DISTRIBUTE base ....

         LOGICAL DeflectFull = .FALSE.
         LOGICAL FeFull      = .FALSE.

      CONTAINS
         SUBROUTINE PutBase(b)
            TYPE (surface), INTENT(IN) :: b
            base = b;  deflected = b
            FeModel =  GenFeModel(b, FeModel)
            FlowSoln = InitSoln(b)
            DeflectFull = .TRUE.
            FeFull      = .TRUE.
         END

         SUBROUTINE PutDeflected(d) WHEN (.NOT. DeflectFull)
            TYPE (surface), INTENT(IN) :: d
            deflected   = d
            DeflectFull = .TRUE.
         END

         SUBROUTINE GetDeflected(d) WHEN (DeflectFull)
            TYPE (surface), INTENT(OUT) :: d
            d = deflected
            DeflectFull = .FALSE.
         END

         SUBROUTINE GetFeModel(f) WHEN (FeFull)
            ...
         SUBROUTINE GetSurfForces(f)
            ...
         SUBROUTINE GetFlow(f)
            ...
         SUBROUTINE PutFlow(f)
            ...
         LOGICAL FUNCTION within_tol(...)
            ...
       END SGeomSDA
```

Figure 3: Surface Geometry SDA

```
      SDA TYPE FeSolverSDA(Surf, Sens, ...)
         SDA (SGeomSDA) Surf
         SDA (SensSDA) Sens
!HPF$ PROCESSORS P(number_of_processors())
          ...
      CONTAINS
         SUBROUTINE Analyze(...)
            converged = .FALSE.
            CALL Surf%GetFeModel(FeModel)
! - discipline optimization loop
            DO WHILE (.NOT. converged)
               CALL Surf%GetSurfForces(forces)
               CALL fesolve(forces, FeModel, deflect, ...)
               CALL Surf%PutDeflected(deflect)
               converged = Surf%within_tol(...)
            END DO

         END

         SUBROUTINE Gradient(...)
            ...
            sens = ...
            CALL Sens%PutFeSens(sens)
         END
      END FeSolverSDA
```

Figure 4: Finite Element Solver

Figure 4 shows the SDA type *FeSolverSDA* which represents the structural analysis code that contains two methods: *Analyze* and *Gradient*. The former carries out the structural analysis on the current geometry while the latter computes the sensitivities with respect to the design variables. As indicated earlier, the internal data parallelism can be expressed using the directives and constructs of HPF. We discuss this further in the next subsection.

4.2.2 Putting it together

The main program, shown in Figure 5, initiates the SDAs and controls the overall optimization cycle. It declares instances of the four SDAs: *FeSolver* and *FlowSolver* representing the discipline codes and *SurfaceGeom* and *Sensitivities* as the data SDAs storing the data values to be shared by the discipline codes. The implicit method *CREATE* is called to allocate and initialize each of the SDA objects. The user may augment the system initialization by defining an *INIT* method which is implicitly called after the call to *CREATE*. Arguments passed in the *CREATE* method are used to initialize the SDA object. Thus, for example, the *SurfaceGeom* and *Sensitivities* are passed in as arguments to the the *FeSolver* and *FlowSolver* SDAs so that the shared data can be accessed by the latter two SDAs.

The *on clauses* associated with the create calls specify the resources to be used for the SDAs. All four SDAs are internally data parallel and use multiple processors for their executions. The two computation SDAs, *FeSolver* and *FlowSolver* are allocated on the machines "XYZ" and "RST" and use four and eight processors respectively. On the other hand, the machine "ABC" is designated as the data server and the two SDAs *SurfaceGeom* and *Sensitivities* use four processors each on it. These processor allocations match up with the HPF processor and distribution directives specified in the respective SDA type definitions. For example, since the SDA *SurfaceGeom* is allocated on four processors, the processor array *P* declared in its type definition (see SDA type *SGeomSDA* as shown in Figure 3) will be instantiated as an array of four processors. That is, for the SDA instance *SurfaceGeom*, the HPF function *number_of_processors()* will return

```
PROGRAM Optimizer
  SDA (FeSolverSDA)    FeSolver
  SDA (FlowSolverSDA) FlowSolver
  SDA (SGeomSDA) SurfaceGeom
  SDA (SensSDA) Sensitivities
    ...
  EVENT e
  TYPE (surface) geom

! – read input arguments and create SDAs
  CALL SurfaceGeom% CREATE (...) ON ( MACHINE="ABC", PROCESSORS =4)
  CALL Sensitivities% CREATE (...) ON ( MACHINE="ABC", PROCESSORS =4)
  CALL FeSolver% CREATE (SurfaceGeom, Sensitivities, ...) ON ( MACHINE="XYZ", PROCESSORS =4)
  CALL FlowSolver% CREATE (SurfaceGeom, Sensitivities,...) ON ( MACHINE="RST", PROCESSORS =8)
! – initialize geometry
  geom = GenBaseGeom(...)

! – optimization loop
  converged = .FALSE.
  DO WHILE (.NOT. converged)
     CALL SurfaceGeom%PutBase(geom)

     e = SPAWN FeSolver%Analyze(...)
     CALL FlowSolver%Analyze(...)
     WAIT (e)

     e = SPAWN FeSolver%Gradient(...)
     CALL FlowSolver%Gradient(...)
     WAIT (e)
     converged = Sensitivities%converged(...)
     IF ( .NOT. converged) geom = ImproveGeom(geom)
  END DO

! – save SDAs if necessary
! – kill all SDAs
END
```

Figure 5: Main program: Optimizer

four. As indicated before, the data within the SDA can now be distributed using the full power of the HPF mapping directives.

Once the SDA object has been created, its public data can be accessed and the associated methods called using a syntax similar to that for derived types in Fortran. As noted before, there are two types of method activations. First is a **synchronous method activation**, in which the caller is blocked until the method call returns. Second is an asynchronous method activation, using the keyword **SPAWN**, in which the caller is not blocked by the method call thus providing support for concurrent activity. For example, in Figure 5, the *Optimizer* uses a spawn statement to invoke the *Analyze* method of the *FeSolver* while using a synchronous method activation for the *Analyze* method of the *FlowSolver* allowing the two activations to execute simultaneously. The spawn statement returns an *event* which can be used in a **WAIT** statement to wait for the completion of the associated method call.

The *Optimizer* controls the outer optimization loop while the *FlowSolver* and *FeSolver* handle the inner optimization cycle for a combined aeroelastic analysis of a given geometry. The Optimizer initiates execution of the inner cycle by storing the initial geometry in the *SurfaceGeom* SDA using the *PutBase* method. *PutBase*, as shown in Figure 3, stores the geometry in the variable *base*, initializes the variable *deflected*, and sets the logical variable *DeflectFull* to *true*. Based on this geometry, it also generates a finite element model, *FeModel*, to be used by the *FeSolver* task and an initial flow solution, *FlowSoln*, for the *FlowSolver* task. The *Optimizer* then calls the analysis methods in the *FlowSolver* and *FeSolver* tasks. Note since the former is activated asynchronously, the two analysis routines are executed in parallel.

The *Analyze* method of the *FeSolver* task, shown in Figure 4, uses the *GetFeModel* method to obtain the finite element model generated on the basis of the current geometry. Similarly, it uses the *GetSurfForces* method to obtain the surface forces generated from the current flow solution. These two data items are used to compute the deflection of the aircraft configuration. The new deflected geometry is then put back into *SurfaceGeom*. Similarly, the *FlowSolver* task (not shown here) acquires the current geometry (using the *GetDeflected* method) and an initial flow solution (using the *GetFlowSoln* method) and produces a new flow solution which it puts back into *SurfaceGeom*.

The inner aeroelastic optimization cycle continues until the deflections are within a specified tolerance limit. At each step of the cycle, the *FeSolver* uses forces based on the current flow solution to produce new deformations, while the *FlowSolver* uses the deflected geometry and the previous flow solution to produce a new solution. Note that the logical variables and the condition clauses in the *SurfaceGeom* SDA are set up to synchronize the parallel tasks. For example, the logical variable *DeflectFull* is used so that the old deflected geometry cannot be replaced by a new one until the old one has been accessed.

After the inner cycle has converged, the *Optimizer* activates the *Gradient* methods of the discipline tasks to generate the sensitivity derivatives with respect to the different design variables. This data is stored in the *Sensitivities* SDA, not shown here, by the discipline tasks. Based on this data and the objective function, the *Optimizer* decides whether to terminate the program or to produce a new base geometry which is then put in *SurfaceGeom* to start a new round of the inner cycle. Once an optimal configuration of the aircraft has been achieved, the SDA data can be saved and the SDAs terminated.

4.3 Opus for MDAs

Above we have given an overview of the features of Opus and explained how they can be used to express a multidisciplinary application. The distinct address spaces associated with SDAs provide independent name spaces for the discipline codes. SDAs also allow the definition of clean interfaces between computational codes, by specifying precisely those data structures that are required for the interaction. This allows a computational code to be easily changed without affecting the other codes as long as the interface is maintained.

The use of SDAs as intermediate data repositories provides the opportunity to carry out any required conversion and translation independent of both the producing and the consuming code. For example, the *FlowSolver* SDA in the above example produces the pressure values on the aero-grid and places them in the *SurfaceGeom* SDA. These pressures can be interpolated to loads on the structural grid within the *SurfaceGeom* SDA before they are accessed by the structural analysis code, *FeSolver*.

Opus relies on the HPF directives to support data parallelism within SDAs. It currently does not provide parallelism across methods within an SDA, since only one method can be active in an SDA object at any time. However, multiple methods belonging to different SDA objects can be active simultaneously, based on the Opus facility for asynchronous method activation. Since the methods can be internally data parallel, this approach cleanly integrates both task and data parallelism within a single language framework.

In reviewing the language requirements necessary for multidisciplinary applications discussed in Section 2, we see that Opus has shortcomings in two areas. First, the current version of Opus has been defined in the context of Fortran and HPF and does not support codes written in other languages. We are currently studying the issues involved in incorporating bindings to other languages, however, we do not anticipate any major roadblocks. Second, the synchronization mechanism in Opus is buried within the method call and the associated *when* condition. There is no mechanism for direct fine-grained signaling between modules. We are currently investigating if such a feature is required.

5 Conclusion

In this paper, we have discussed language requirements for multidisciplinary applications, based on a study of their characteristic properties. The major areas where language support is needed include modularity, process generation and synchronization, as well as efficient access to shared data sets.

Many existing parallel languages – such as Linda, PCF, or HPF – fall short with respect to at least one of these categories and thus cannot provide the required performance and/or functionality.

Although none of the presently existing approaches can fully satisfy all requirements we believe that the combination of an object-based approach – which provides flexibility and elegance – with coarse-grain task parallelism and HPF-style data parallelism – which can be used to achieve highly efficient data parallel operations – may well be the most promising approach to follow in the future.

Acknowledgements We are thankful to the referees whose comments helped improve the presentation in the paper.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19:26–34, August 1986.

[2] American National Standard Committee X3H5. Parallel Processing Model for High Level Programming Languages. Draft Proposed American National Standard for Information Processing Systems, April 1995.

[3] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):3–44, March 1983.

[4] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[5] R. Bisiani, and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[6] K. M. Chandy and C. Kesselman. *CC++: A Declarative Concurrent Object-Oriented Programming Notation.* In Research Directions in Concurrent Object-Oriented Programming. MIT Press, 1993.

[7] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[8] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima. Opus: A Coordination Language for Multidisciplinary Applications. To be published in *Scientific Programming*, 1997 (Also available as ICASE Report No. 97-30 ICASE, NASA Langley Research Center, Hampton VA.)

[9] I. T. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 25(1), 1995.

[10] High Performance FORTRAN Forum. *High Performance FORTRAN Language Specification, Version 2.0*, Technical Report, Computer Science Department, Rice University, Houston, TX, January 1997 (HPF Version 1.1 available as Scientific Programming, Vol. 2, no. 1-2 (Spring and Summer 1993), pp. 1-170, John Wiley and Sons.)

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.0* Technical Report, Computer Science Department, University of Tennessee, Knoxville, Tennessee, 1997.

[12] OpenMP Consortium: OpenMP Fortran Application Program Interface, Version 1.0, October, 1997 (http://www.openmp.org).