A Software Architecture for Multidisciplinary Applications: Integrating Task and Data Parallelism*

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

[a]Institute for Software Technology and Parallel Systems,
University of Vienna, Brünner Strasse 72, A-1210 Vienna AUSTRIA
E-Mail: zima@par.univie.ac.at

[b]ICASE, MS 132C, NASA Langley Research Center, Hampton VA. 23681 USA
E-Mail: pm@icase.edu

Abstract

Data parallel languages such as Vienna Fortran and HPF can be successfully applied to a wide range of numerical applications. However, many advanced scientific and engineering applications are of a multidisciplinary and heterogeneous nature and thus do not fit well into the data parallel paradigm. In this paper we present new Fortran 90 language extensions to fill this gap. Tasks can be spawned as asynchronous activities in a homogeneous or heterogeneous computing environment; they interact by sharing access to Shared Data Abstractions (SDAs). SDAs are an extension of Fortran 90 modules, representing a pool of common data, together with a set of methods for controlled access to these data and a mechanism for providing persistent storage. Our language supports the integration of data and task parallelism as well as nested task parallelism and thus can be used to express multidisciplinary applications in a natural and efficient way.

*The work described in this paper was partially supported by the Austrian Research Foundation (FWF Grant P8989-PHY) and by the Austrian Ministry for Science and Research (BMWF Grant GZ 308.9281- IV/3/93). This research was also supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

1 Introduction

Data parallel languages, such as High Performance Fortran (HPF) [?] and Vienna Fortran [?, ?], are maturing and can readily express the parallelism in a broad spectrum of scientific applications. In this sense, data parallel languages have proven highly successful.

However, scientific and engineering applications are a moving target. With the anticipated arrival of teraflop architectures, the complexity of simulations being tackled by scientists and engineers is increasing exponentially. Many of the new applications are multidisciplinary: programs formed by pasting together modules from a variety of related scientific disciplines. Such multidisciplinary programs raise a host of complex software integration issues, in addition to parallel performance issues. HPF, and its siblings, are completely inadequate for this class of applications.

Environmental simulation is one area in which such applications are beginning to arise. One might wish to couple a variety of environmental models, each given initially as separate programs:

1. A plant biology model for the Florida Everglades
2. A model of the gulf stream dynamics
3. A climate model for North America
4. A solar radiation model

The goal is then to interconnect these disparate models into a single multidisciplinary model subsuming the original models and their interactions. At the same time, the parallelism both within and between the discipline models needs to be exposed and effectively exploited.

Precisely analogous issues arise in multidisciplinary optimization. In designing a modern aircraft, for example, one has a wide variety of interacting disciplines: aerodynamics, propulsion, structural analysis and design, controls, and so forth. An optimal engineering design is necessarily an admixture of suboptimal designs in each discipline. The essential goal is to correctly couple a sequence of complex scientific and engineering programs from different disciplines, each designed and implemented by different groups, into a coherent whole capable of effective multidisciplinary optimization. Moreover, the collection of programs chosen must remain flexible, since the choice of programs tends to evolve rapidly as the simulation methodology changes, or as unanticipated interactions force alteration of the mix of disciplines or programs being used.

In attempting to carry out such multidisciplinary design, scientists are confronted with a host of complex software engineering issues, together with the necessity of effectively mapping the resulting unwieldy codes to a heterogeneous network of workstations and massively parallel architectures. In this environment, statically forming a "task graph" and coupling tasks via message plumbing appears virtually unworkable. In a message-passing environment, the design of each task requires

intimate knowledge of the behavior of all coupled tasks. Given a rapidly evolving mix of program modules, as will occur in many multidisciplinary applications, a more flexible software environment appears critical.

Our approach is designed to address this problem. It provides a software layer on top of data parallel languages, designed to address both the "programming in the large" issues, and the parallel performance issues arising in complex multidisciplinary applications. A program executes as a system of *tasks* which interact by sharing access to a set of *Shared Data Abstractions (SDAs)*. SDAs generalize Fortran 90 modules by including features from both *objects* in object-oriented data bases and *monitors* in shared memory languages. The idea is to provide persistent shared "objects" for communication and synchronization between large grained parallel tasks, at a much higher level than simple communication channels transferring bytes between tasks.

Tasks in our system are asynchronously executing autonomous activities to which resources of the system are allocated. They may embody nested parallelism, for example by executing a data parallel HPF program, or by coordinating a set of threads performing different functions on a shared data set. Moreover, the system of tasks associated with an application may execute in a homogeneous or heterogeneous environment.

A set of tasks may *share* a pool of common data by creating an SDA of appropriate type, and making that SDA accessible 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. The combination of the task and SDA concepts should form 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.

The structure of this paper is as follows. The next section provides an overview of task management, while Section **??** presents the data abstractions required for sharing data between the tasks. Section **??** describes a multidisciplinary application, the optimal design of an aircraft, and shows how it would be programmed using the language features described in this paper. This is followed by a section on related work and a brief set of conclusions.

2 Tasks

Tasks are *spawned* by explicit activation of *task programs*. A task program is syntactically similar to a Fortran subroutine (except for the keyword **TASK CODE** which is used instead of **SUBROUTINE**) but has a different semantics: different tasks execute asynchronously and independently as long as they are not synchronized. A task *terminates* if its execution reaches the end of the associated task program code, or if it is explicitly *killed*. A task *exists* during its *lifetime*,

which is the period of time between spawning and termination.

The interface between a task and its environment is defined by the arguments passed to the task and the structure of the associated SDAs. All arguments of a task except for status variables must have intent IN. Common blocks and modules **cannot** be shared between tasks: in particular, the spawning of a task creates a task-specific instance of every common block in the task program, and a task has no access to objects belonging to a common block associated with its parent. The semantics of modules is defined similarly.

Tasks are units of *coarse-grain parallelism* executing in their own address space and operating on a set of system resources *allocated* to them at the time of their spawning, such as machines and their associated processors, memory modules, and file space. The spawning statement may contain an explicit resource request – it is then the system's responsibility to allocate sufficient resources to satisfy this request – or it may let the system decide the resource requirements.

2.1 Task Spawning

A task is created by executing a *spawn-statement*. The spawn statement identifies the task program to be executed, together with an optional argument list and resource request:

SPAWN *taskprogram-name* ["(" *argument-list*")"] [**ON** *resource-request*]

The execution of a spawn statement

- creates a new task,
- passes a list of arguments to the task,
- allocates resources to the task,
- returns a unique integer value, the *task identification**, and
- initiates the execution of the task program.

The task in which the spawn statement is executed is called the *parent* of the newly created task. The intrinsic function **SELF** yields the identification of the executing task.

The argument list may specify status variables that provide the user with information concerning the success or failure of the spawning operation. If a spawn-statement fails (for example, because its resource request cannot be met), its effect is empty, except for possible implicit assignments to status variables which indicate the cause of the failure by returning an error code.

All other arguments specified in a spawn statement must be of intent IN.

In the following, we will assume that the spawn statement is executed successfully, if nothing to the contrary is said. The newly-created task will be denoted by T.

*This value can be assigned to an integer variable and used in *task expressions* (see Section **??**) to gain access to the task.

2.1.1 Resource Specification

Each task operates on a set of resources which are allocated at the time the task is created, and deallocated at the time of its termination. Different tasks may execute on disjoint or overlapping sets of resources.

If a *resource-request* is specified in a spawn statement, then it determines a set of resources that must be allocated necessarily to the newly created task. In the absence of such a request the system allocates resources it deems necessary to execute the task.

A resource request may specify the physical machine on which the task is to be executed, along with additional requirements related to this machine. It is structured as follows:

[**MACHINE** "(" *physical-machine-spec* ")"] M ["," **PROCESSORS** "(" *processor-spec* ")"] [*other-resource-spec*]...

The *physical-machine-spec* can be given either directly or indirectly:

- A *direct specification* identifies a physical machine by a string with a system-dependent meaning, for example,

  MACHINE ('TOUCHSTONE DELTA...')

  The concept of *machine* that we use here allows a broad interpretation: for example, it may denote a specific vector machine, a workstation, a parallel architecture, a cluster of workstations, or any of their components that can be used for the independent execution of programs. It may also denote a class of machines with the system being free to choose any specific machine from the class.

- An *indirect specification*, for example **MACHINE** (TT), provides the identification of a task (which must exist) or the name of an SDA (which must have been initialized). In this case, the physical machine is the same as the machine allocated to TT.

For the following, assume that M is the machine on which task T is to be executed. Any additional resource requirements specified in the spawn statement refer to components of M. We will actually restrict our discussion here to the processor specification *processor-spec*, which identifies the processor set to be associated with T. Other requirements, such as those for main memory or file space, may have to be satisfied to render the spawn successful.

The processor set can be specified indirectly via a task identification or SDA name, with analogous semantics as before. For a *direct* specification, the following options exist:

- A *processor reference*. In this case, *processor reference* identifies a processor section of M, which must be associated with the parent of T.

- An integer expression, yielding a value k identifying the *number* of processors on machine M that are needed for the execution of the task.

  If the expression is preceded by **NEW** and M was obtained by an indirect specification referring to TT, then k "new" processors – in addition to those already associated with TT – have to be allocated.

If any of the potential components of a resource request is missing, a system-dependent decision is made.

We conclude this section with a note on the interface to Vienna Fortran and HPF procedures. If T is spawned using a Vienna Fortran procedure, say Q, that contains a processor declaration with a symbolic variable name in a *dimension bound expression* – for example, **PROCESSORS** *R(M,N)* – then these variables (M and N in the example) must be dummy arguments of Q and explicitly supplied with proper actual arguments in the spawn statement.

The value respectively yielded by the functions *$NP* in Vienna Fortran and *NUMBER_OF_PROCESSORS* in HPF is determined by the number of processors allocated to T.

Examples

- *T1* = **SPAWN** *Q(K,L,* **STAT** = *RS1*)

  A task is spawned by activating the task program Q with arguments K and L. The task is executed on a system-defined machine and processor set. The execution of spawn yields an integer value for the identification of the task which is assigned to the integer variable *T1*. Status information regarding the execution of the task is returned in variable *RS1*.

- *T2* = **SPAWN** *Q(K+1,L+1)* **ON MACHINE** *('Intel iPSC860/64...')*
  Similar to above, but here the machine on which the task is to be executed is specified explicitly. The number of processors allocated to the task is determined by the system.

- *T3* = **SPAWN** *Q(K+1,L-1)* **ON MACHINE** (*T2*), **PROCESSORS** *(32)*
  This task is executed on the same machine as *T2*; it requires 32 processors (which may or may not coincide with the processors allocated to *T2*).

- *T4* = **SPAWN** *Q(K,L)* **MACHINE** (**SELF**), **PROCESSORS** (**SELF**)
  This task is executed on the same machine and processor set as its parent.

- *T5* = **SPAWN** *QQ(K-1,L,8,4)* **ON MACHINE** (*T2*), **PROCESSORS**(**NEW** 32)
  Similar to the last example, but in this case the task requires 32 processors *in addition* to those already allocated to *T2*.

- *T6* = **SPAWN** *QQ(K-1,L,8,4)* **ON MACHINE** (*T2*), **PROCESSORS** (32)
  This task is executed on the same machine as *T2*. Assuming that *QQ* contains a processor declaration of the form **PROCESSORS** *R(M,N)* and that the last two dummy arguments of *QQ* are *M* and *N*, then the corresponding actual arguments determine the shape of *R*.

- *T7* = **SPAWN** *QQ(1,2,4,8)* **ON MACHINE SELF**, **PROCESSORS** *RR(N1:N2,M1:M2,K1:K2)*
  This task is executed on the same machine as its parent and requests the processor array section *RR(N1:N2,M1:M2,K1:K2)* to be allocated, where *RR* is a three-dimensional processor array associated with the parent.

2.2 Task Termination

A task *terminates* if the execution of the associated subroutine comes to its end, or if its execution is explicitly ended by a *terminate statement*. If a task terminates, then all its children terminate as well.

The terminate statement has the form

TERMINATE [*task-expression-list*]

A *task expression* is an integer expression whose value identifies an existing task. This statement terminates all tasks specified in the task expression list. The keyword **CHILDREN** identifies the set of all children that were spawned by the executing task and still exist.

If the list is empty, the task executing the statement is terminated.

Examples

TERMINATE T2, T3

This terminates *T2* and *T3*.

TERMINATE

This terminates the task executing this statement (and its children).

TERMINATE CHILDREN

This terminates only the children of the task executing this statement.

2.3 Task Coordination

Tasks are coordinated by accessing methods in SDA objects. One basic mechanism provided in the language is the *condition clause*, which is a boolean guard attached to a public method of an SDA. This method can then be executed only when the evaluation of the boolean expression yields **true**; if necessary, it is blocked until the condition is satisfied (see Section **??**).

Another mechanism is synchronization depending on task termination: **WAIT** $tex_1, \ldots, tex_n$, where the tex_i are task expressions, blocks the executing task until *all* tasks associated with the tex_i have terminated. If the list of task expressions is preceded by **ANY : WAIT ANY** $tex_1, \ldots, tex_n$, then the executing task waits until *any one* of the tasks associated with the tex_i terminates.

Other mechanisms for more sophisticated coordination, including a low-level event-based facility are currently under investigation and will be added to the language at a later point.

Example

Assume that $Q1$, $Q2$, and $Q3$ are task programs. Then

TT1 = **SPAWN** *Q1(...)*
TT2 = **SPAWN** *Q2(...)*
TT3 = **SPAWN** *Q3(...)*
WAIT CHILDREN

causes the executing task to initiate the tasks *TT1*, *TT2*, and *TT3*, and then wait for the completion of all three tasks (we assume that no other children exist). This has an effect similar to the parbegin-parend construct used in other languages [**?**]:
PARBEGIN *Q1(...)*, *Q2(...)*, *Q3(...)* **PAREND**

3 Shared Data Abstractions

Tasks, as described in the last section, share information using **Shared Data Abstractions (SDAs)**. SDAs can be persistent in the sense that they allow program data to be stored in external storage in a structured way rather than as just a sequence of bytes.

In the following, we distinguish between an *SDA type* which is a type specification for an SDA and the *SDA object* itself. The latter refers to an *instance* of an SDA type. We also distinguish between an SDA object and an *SDA variable* which is an internal program name which denotes the SDA. A specific SDA object may have different *internal* names, e.g., in different tasks. However, if an SDA object has been stored externally it will acquire a unique *external* name. We use the term SDA for all three concepts interchangeably if the meaning is clear from the context.

An SDA consists of a set of data structures along with the methods (procedures) which manipulate this data. Tasks can share an SDA object and can asynchronously call the associated methods. However, *each call to the SDA has exclusive access to the data in the instance.* That is, only one method call associated with an SDA object can be active at one time. Other request are queued and the calling tasks blocked until the currently executing method completes its execution. The execution of individual methods can also be controlled by the use of a *condition clause* as described below.

3.1 Specification of SDA Types

The SDA type specification syntax, modeled after the Fortran 90 module syntax, contains two parts. The specification part consists of all the declarations, including types and variables, while the subprogram part specifies the subprograms associated with the SDA type. As in a Fortran 90 module, each subprogram declared within an SDA type has access to all the entities declared in the SDA type through host association. The SDA type specification extends the Fortran 90 module specification in several ways, as described in the following subsections.

SDA arguments

The SDA type header consists of the SDA type name along with a list of dummy arguments similar to those of any Fortran 90 procedure. These arguments can be used to parameterize the internal data structures of the SDA (including local arrays) The arguments of an SDA must be of intent IN.

The SDA type header can also include an optional *of-clause* which is used to specify a special argument, a *type-name*. This allows a type to be passed in as an argument to the SDA which can then be used as a type specification within the SDA specification.

For example, the following code fragment represents the specification part of an SDA type which provides a stack for communicating data between tasks:

```
SDA TYPE  stack (max) OF (T)
   INTEGER max
   TYPE(T) :: lifo(max)
   INTEGER count
    ...
CONTAINS
    ...
END stack
```

Here, *max* is an integer argument which specifies the maximum size of the stack whereas T is a type argument which allows *lifo* to be declared as an array of type T. Thus, as shown in section **??**, the same SDA type specification can be used to declare a stack of integers, a stack of reals, etc. The name T designates a type and the only operations allowed on objects of type T are: assignment, checking for equality and passing them as arguments to methods.

Accessibility of SDA entities

As in the case of a Fortran 90 module, the entities declared inside the SDA type are considered public unless explicitly declared to be private using the keyword PRIVATE. The default can be changed by a PRIVATE statement with an empty entity list. Then all entities are private unless explicitly declared to be public using the keyword PUBLIC.

Note that public variables of the SDA are directly visible and accessible to all tasks having access to the SDA. However, as in the case of method calls, access to these variables is an atomic operation, and the task accessing the variable has exclusive access to the whole SDA during the operation.

SDA Methods

Public methods may be called by tasks having access to the SDA. Each public method can have an associated *condition clause* which consists of a logical expression. The logical expression controls the execution of the method, i.e., a call to the method is blocked until the logical expression evaluates to **true**. The logical expression can be constructed using the entities declared in the specification part of the SDA type along with the dummy arguments of the associated method. However, the expression is restricted in that its evaluation is not allowed to have any side effects which change the state of the SDA.

The condition clause is attached to the header of the procedure in the subprogram specification part, as shown in the following code fragment:

```
SDA TYPE  stack (max) OF (T)
   INTEGER max
   TYPE(T), PRIVATE :: lifo(max)
   INTEGER, PRIVATE count

    ...
CONTAINS
   SUBROUTINE get (x) WHEN (count .gt. 0)
      TYPE(T) x
```

```
      x = lifo(count)
      count = count - 1
    end

    SUBROUTINE put (x) WHEN (count .lt. max)
      TYPE(T) x
      count = count + 1
      lifo(count) = x
    END

    INTEGER FUNCTION cur_count
      cur_count = count
    END
     ...
  END stack
```

Thus, in the above code fragment, *lifo* and *count* are private whereas the methods *cur_count, put* and *get* are public. The method *cur_count* does not have an associated condition clause and hence can be executed whenever it has exclusive access to the SDA. However, as specified in the condition block, the subroutine *get* can only be executed if *count* is greater than zero. Similarly, the subroutine *put* can only be executed if *count* is less than *max*.

A public method cannot directly or indirectly call any other public method associated with the same SDA.

Each SDA has three implicit public methods: *INIT, LOAD* and *SAVE*. The first two are used to initialize an SDA while the third is used for saving the current state of the SDA to external storage for later use. The three methods are described in Section **??**.

Distribution of Data

Each SDA may have an optional processors statement, as for example HPF or Vienna Fortran procedures, which allows the internal data structures of the SDA to be distributed across these processors. The dummy arguments of the SDA methods can be distributed using the rules applicable to any HPF procedure.

3.2 SDA Declaration and Use

An SDA type name can be used to declare SDA variable names of the type in a manner similar to that used for Fortran 90 derived type definitions. The declaration consists of the name for the

SDA along with an *of-clause* if required by the specification. The following code fragment declares two objects of type *stack* (see Section **??**):

SDA (stack) **OF** **INTEGER** :: int_stack

SDA (stack) **OF** **TYPE** (user_type) :: user_stack

The of-clause provides a type name to be associated with the type argument of the SDA type. Thus, *int_stack* denotes an SDA which manipulates integers while *user_stack* will manipulate objects of a user defined type, *user_type*.

The declaration statements create SDA variable names of the specified type in an *uninitialized* state. The SDA name must be initialized by associating it with an SDA object before it can be used. This can be done using the *INIT* or *LOAD* methods, as shown below. Only the task declaring an SDA variable can initialize the variable. An SDA name and the SDA object it denotes exists as long as the program unit declaring it is active. The object can be made *persistent* by calling the *SAVE* method to transfer the SDA data to external storage.

An SDA variable declaration is not allowed to have the **POINTER** or **ALLOCATABLE** attributes. Conceptually an SDA variable is a pointer to an SDA object. As a consequence, all tasks to which an SDA object is passed have access to the same copy of the object and hence can communicate with each other using the object[†].

An SDA can be passed as argument to procedures within a task and also to other tasks as they are being spawned.

Entities declared in an SDA type specification are invoked using the same syntax as used for derived type. Thus, *int_stack%max* accesses the value of the *max* variable associated with the SDA *int_stack*. SDA methods can be invoked using a similar syntax:

sda-name%method-name "(" *arg-list* ["," **STAT** = *stat-variable*] ")"

where *sda-name* is the name of the SDA object, *method-name* is the name of the method being invoked and *arg-list* is the list of arguments required by the method. With any SDA method call, the user can supply an optional status variable, preceded by the specifier **STAT**=. The variable is set to a non-zero value if the method call fails for any reason (see generic SDAs defined later in this subsection).

As noted before, each SDA has three implicit public methods: *INIT*, *LOAD* and *SAVE*. The first two are used to initialize an SDA name while the last method saves the current state of the SDA in external storage.

[†]Note that this does not conflict with the requirement that all task arguments be of intent IN. The SDA variable that is passed is intent IN, i.e., its value cannot be changed. However, method calls to the object pointed to by the variable can change the state of the object.

Initializing an SDA variable

INIT Method: The *INIT* method is used to initialize an SDA variable. It is called using the input arguments as specified in the SDA type specification. The method creates an instance of the SDA by allocating the required data structures and performing the default initialization. Thus, the following call,

CALL int_stack%INIT(100, STAT = init_status)

initializes the *int_stack* SDA to be of size 100. Again, the **STAT** variable *init_status* is set to a non-zero value if the initialization fails for any reason, e.g., if there is not enough memory to allocate the data structures.

An optional *resource-request* (as described in Section **??** for task spawning) allows the user to specify resources to be used for the SDA. The user can also provide a method called *INIT* in the SDA type specification which includes code for initializing the internal data structures of the SDA. This code is executed after the data structures for the instance have been allocated.

LOAD Method: The *LOAD* method call is used to "load" an SDA object with data which had been "saved" earlier using the *SAVE* method. Each call to the *LOAD* method makes an internal copy of the external data, leaving the external data untouched. The *LOAD* call takes a string (constant or variable) as argument which identifies a saved SDA. For example, in the following statement, data saved using the external name *stack_sav* is loaded into the SDA object, *user_stack*.

CALL user_stack%LOAD('stack_sav' , STAT = init_status)

First, space for the internal data structures of the object is allocated, and then the data from the saved SDA is loaded into the SDA object. As in the case of *INIT*, an optional *resource-request* allows the user the specification of resources to be used for the SDA object. Note that the type of the SDA object must match the type of the saved object. Two SDA types are considered equivalent if a) the public variables of the SDA types are equivalent in the same sense as the fields of two Fortran 90 derived types are equivalent, and b) the method names and arguments of the public methods of the two types are the same.

Saving an SDA object

The *SAVE* method allows the user to save the state of the SDA on external storage for later reuse. The method takes a string (constant or variable) as an optional argument which is used as an external name for the saved object. The following statement saves the current state of *user_stack* using the external name *'stack_sav'*.

CALL user_stack%SAVE('stack_sav' , STAT = sav_status)

If the external name denotes a currently saved object it is overwritten with the new state; otherwise a new saved object is created. If the variable name had been initialized using a *LOAD* call then the string argument may be omitted. In this case, the external name used for the load is used for the save, overwriting the original data.

Generic SDA variables

The language allows the declaration of generic SDA variables whose type is determined by the data saved on external storage. Thus, the declaration

SDA :: gen_sda

specifies that *gen_sda* is an SDA name which will be associated with and SDA object of an unnamed type. Such an variable can only be initialized using the *LOAD* method and thus inherits the type of the loaded object. Note that using this facility implies runtime checks to determine whether a method called with such an object exists and, if it does, whether the argument types match. However, a judicious use of status variables provides a graceful failure mode.

4 Example

In this section we describe, in relative detail, an example of an application expressed in our language. The example chosen is the simultaneous optimization of the aerodynamic and structural design of an aircraft configuration. By the standards of multidisciplinary optimization (MDO) this is a comparatively simple example involving just two disciplines. However, it does illustrate some of the capabilities of our system, as well as show some of the software complexity of this class of applications, and also the potential for task level parallelism.

The structure of this program is shown in Figure **??**. Here rectangles represent tasks, while ovals represent SDAs. Execution begins with the routine *Optimizer*, shown in Figure **??**, which creates the three SDAs shown, then spawns the other three tasks shown.

The functions of the three spawned tasks are as follows:

1. *GridGen:* the grid generator which takes the current geometry (aircraft configuration) and produces a three-dimensional aerodynamics grid surrounding it, for use in the flow solver.

2. *FlowSolver:* the flow solver which, beginning with the previous flow solution, computes a new solution on the current aerodynamics grid.

3. *FeSolver:* the finite element solver which applies forces corresponding to the current flow solution to the structure, to determine new structural defections.

In the simple variant of this optimization program shown, only one of these three tasks is active at a time, with control flow passing sequentially between tasks. There are however, a number of alternatives having tasking-level parallelism, as discussed at the end of this section.

Each of the tasks takes data from one or more SDAs, performs a sequence of computations on it, then inserts the results into one or more other SDAs. For example, the grid generator takes as input the current surface geometry, which is field *deflected* in SDA *SurfaceGeom*. It then computes with this data, producing a new aerodynamics grid, which it inserts into SDA *AeroGrid*. Similarly, the *flow solver* uses the current grid and previous solution in *AeroGrid* to produce a new flow solution put in *AeroGrid*. The structure of the grid generator code is shown in Figure **??**; we omit the code for the flow-solver.

The structure of the two SDAs used here is shown in Figures **??** and **??**. The *SurfaceGeom* SDA contains the method *GetFeModel*, which returns a new finite element model for the aircraft. We could have created a separate task *finite element model* to do this, but in this case, generating the finite element model is trivial, so it can simply be a method in the *SurfaceGeom* SDA.

Analogously, *AeroGrid* contains the method, *SurfaceForces*, which computes the pressure loads and viscous stresses acting on the aircraft surface. Logically, one could think of this as either a filter operating on the output of the flow solver, or as a part of the flow solver. However, the former viewpoint is perhaps more natural, since the operation of extracting surface forces is the same, independent of the flow solver used or the use being made of the the surface forces.

The third spawned task is the finite element solver, shown in Figure **??**. This task uses the surface forces in the AeroGrid, together with the finite element model in *SurfaceGeom* to compute new deflections of the aircraft configuration. It also computes the change between the new deflections and previous deflections, which it inserts in the SDA *StatusRecord*.

The SDA *StatusRecord* is shown in Figure **??**. It is used to keep track of the current status of the optimization process, the current drag prediction, and so forth. Control flow circulates in the inner loop of *FeSolver*, *GridGen*, *FlowSolver* until the convergence criterion is met. At this point, the *FeSolver* set the *Done* variable in the *Status* SDA allowing the *Optimizer* to take control. The latter then decides whether to terminate the program or to produce a new base geometry which when put in *SurfaceGeom* starts a new round of the inner loop.

5 Related Work

Task management has been a topic of research for several decades, particularly in the operating systems research community. A good survey of the issues can be found in [**?**]. However, there has not been much attention given to the mechanisms required for managing control parallel tasks, which may themselves be data parallel. In this section we discuss some of these approaches.

Fortran M [?] extends Fortran 77 with a set of features that support message-passing, according to a strictly enforced discipline. *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[‡]. Communication is performed by sending and receiving from ports. Processes are activated by executing a process block – a PARBEGIN/PAREND like construct – or by creating multiple instances in a process loop. The language has constructs for controlling the location of process executions and distributing data in an HPF-like manner. By imposing a FIFO discipline on message queues and guaranteeing a sequential semantics for output arguments determinism is enforced.

Fortran M can be used to create and coordinate threads in a clean and structured way. However, the relatively low level of abstraction associated with the message-passing paradigm, together with the structure imposed on the use of channels and ports for the sake of achieving determinism sometimes leads to difficulties expressing simple and useful communication structures. Such examples include producer-consumer problems with multiple producers and consumers accessing a bounded buffer, or the variants of the readers-writers problem.

The **Fx** Fortran language extensions developed at CMU [?, ?] include *parallel sections* that allow the concurrent activation of subroutines as *tasks*. Tasks communicate by sharing arguments. Arguments can be passed to a task at the time of its activation, or received from a task when it terminates. Each call that activates a task must be accompanied by *input* and *output* directives that specify the shared objects. This provides the compiler with complete information on the required communication.

Fx is well suited to an environment where tasks need to communicate only at the time of spawning and termination, and where nested task-parallelism is not required. If tasks must communicate during their execution, subroutines may have to be split at synchronization points to obtain smaller program units that fit into this scheme. Moreover, this would clearly induce task-spawning overhead.

LINDA [?] provides a virtual shared *tuple space*, to which read and write operations can be applied. It represents a simple and easily usable parallel programming paradigm. However, LINDA lacks the modularity that is required for structuring multidisciplinary applications, and does not allow sufficient control of task execution and resource allocation.

SVM Fortran [?] is a set of extensions for Fortran 77 intended to program shared virtual memory systems. among a large number of features, it provides support for fine-grained control parallelism in a shared memory paradigm along with mechanisms to synchronize and coordinate these tasks.

Other approaches which provide support for managing task parallelism at a high level include

[‡]In addition, many-to-one communication can be expressed.

occam [?], PVM [?], CC++ [?] and Strand [?]. Most of these approaches do not address the issue of integrating task and data parallelism.

6 Conclusion

Complex scientific applications, such as multidisciplinary optimization, provide opportunities for exploiting multiple levels of parallelism; however, they also raise complex programming issues. In this paper, we have presented language extensions which not only allow the specification of parallelism but also provide support for software engineering issues which arise when integrating codes from individual disciplines into a single working application. The user has to explicitly specify tasks and manage concurrent tasks. We presume that data parallelism within these tasks will be specified using an HPF-like approach. The user controls the sharing of information between these tasks through Shared Data Abstractions, which allow the task interfaces to remain independent of each other.

We are in the process of building a prototype implementation and will report the results of these efforts in future papers.

References

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

[2] R. Berrendorf, M. Gerndt, W. Nagel and J. Prümmer. SVM Fortran. Technical Report, Research Center Juelich(KFA), Germany, September 1993.

[3] N. Carriero and D. Gelernter. How To Write Parallel Programs. MIT Press, 1990.

[4] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. Technical Report CS-TR-92-01, California Institute of Technology, 1992.

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

[6] I. T. Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. Technical Report MCS-P327-0992 Revision 1. Mathematics and Computer Science Division, Argonne National Laboratory, June 1993.

[7] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[8] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Scientific Programming 2(1-2):1-170,Spring and Summer 1993.

[9] D. Pountain. *A Tutorial Introduction to Occam Programming.* Inmos, Colorado Springs, Co., 1986.

[10] J. Subhlok,J. Stichnoth,D. O'Hallaron,and T. Gross. Exploiting Task and Data Parallelism on a Multicomputer. Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93).

[11] J. Subhlok and T. Gross. Task Parallel Programming in Fx. Technical Report CMU-CS-94-112, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[12] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. Concurrency: Practice and Experience 2: 315-339 (1990).

[13] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. ICASE Internal Report 21, ICASE, Hampton, VA, 1992.

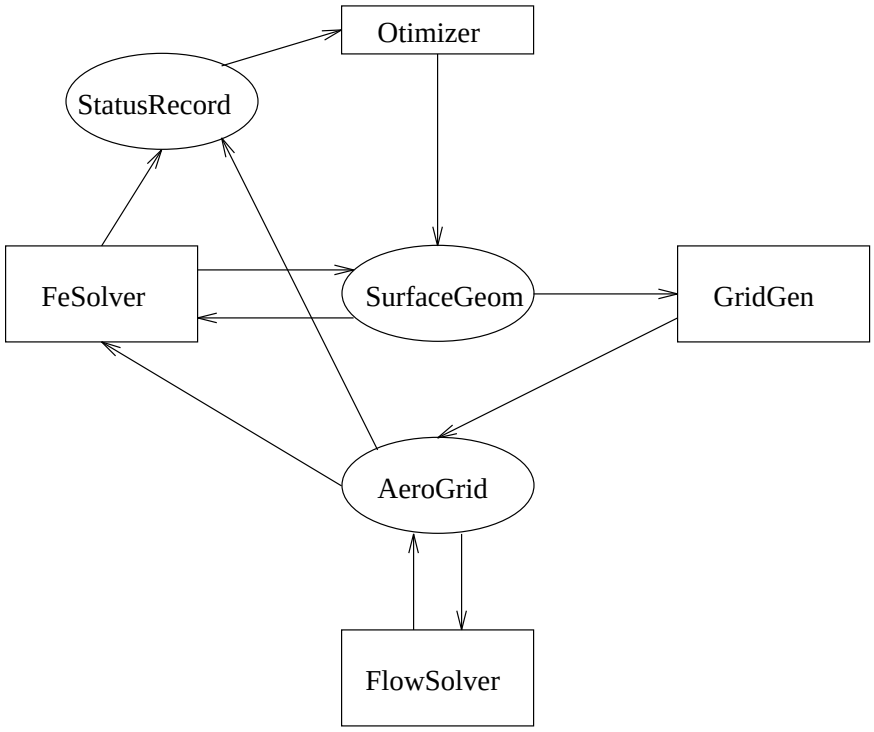

Figure 1: MDO Application

```
      PROGRAM Optimizer
         SDA (SurfaceGeom) Surf
         SDA (AeroGrid)    Grid
         SDA (StatusRecord) Status
         TYPE (surface) geom

! – read input arguments and initialize SDAs
         CALL Surf%INIT
         CALL Grid%INIT
         CALL Status%INIT

! – spawn tasks
         SPAWN FeSolver  (Surf, Grid, Status, ...)
         SPAWN GridGen   (Surf, Grid, ...)
         SPAWN FlowSolver(Grid, ...)

! – initialize  geometry
         geom = GenBaseGeom(...)
         CALL Surf%PutBase(geom)

! – outer loop
         CALL Status          Drag     = Status          DragDiff = Drag
         DO  WHILE (DragDiff .gt. Epsilon)
            geom = ImproveGeom(geom)
            CALL Surf%PutBase(geom)
            CALL Status%GetDone
            OldDrag  = Drag
            Drag     = Status%drag
            DragDiff = Drag-OldDrag
         END  DO

! – save SDAs if necessary

! – kill all tasks
         TERMINATE

      STOP
      END
```

Figure 2: Main program

```
TASK CODE GridGen(Surf, GridSDA, ...)
   SDA (SurfaceGeom) Surf
   SDA (AeroGrid)    GridSDA
   TYPE (surface) geom
   TYPE (FlowGrid) grid

   DO WHILE (.TRUE.)
      CALL Surf%GetDeflected(geom)
      grid = GenAeroGrid(geom)
      CALL GridSDA%Putgrid(grid)
   END DO
END GridGen
```

Figure 3: Grid generator

```
SDA TYPE SurfaceGeom
   TYPE(surface) base
   TYPE(surface) deflected
   TYPE(fe) FeModel

   LOGICAL DeflectFull = .FALSE.
   LOGICAL FeFull      = .FALSE.
   PRIVATE base, deflected, FeModel,DeflectFull, FeFull

CONTAINS
   SUBROUTINE PutBase(b)
      TYPE(surface) b
      base = deflected = b
      CALL GenFeModel(b, FeModel)
      DeflectFull = .TRUE.
      FeFull      = .TRUE.
   END

   SUBROUTINE PutDeflected(d) WHEN .NOT. DeflectFull
      TYPE(surface) d
      DeflectFull = .TRUE.
      deflected   = d
   END

   SUBROUTINE GetDeflected(d) WHEN DeflectFull
      TYPE(surface) d
      DeflectFull = .FALSE.
      d = deflected
   END

   SUBROUTINE GetFeModel(f) WHEN FeFull
      TYPE(fe) f
      f = FeModel
      FeFull = .FALSE.
   END
   ...
END SurfaceGeom
```

Figure 4: Surface Geometry SDA

```
SDA TYPE AeroGrid ( s )
   SDA (StatusRecord) s
   TYPE (FlowGrid) grid
   TYPE (FlowSoln) solution

   LOGICAL GridFull = .false.
   LOGICAL NewFlow = .false.
   PRIVATE grid, solution, GridFull, NewFlow

CONTAINS
   SUBROUTINE init
! – code to initialize solution
   END init

   SUBROUTINE PutFlow(s)
      TYPE (FlowSoln) s
      solution    = s
      NewFlow = .TRUE.
   END

   SUBROUTINE GetFlow(s)
      TYPE (FlowSoln) s
      s = solution
   END

   SUBROUTINE GetSurfForces(f) WHEN NewFlow
      TYPE (SurfForces) f
      REAL drag
      f       = GenForces(FlowSoln)
      drag    = SurfIntegral(f)
      s%drag = drag
   END

   SUBROUTINE PutGrid(g) WHEN .NOT. Gridfull
     ...
   END
   SUBROUTINE GetGrid(g) WHEN GridFull
     ...
   END

     ...
END AeroGrid
```

Figure 5: AeroGrid SDA

```
TASK CODE FeSolver(Surf, GridSDA, Status, ...)
   SDA (SurfaceGeom) Surf
   SDA (AeroGrid)    GridSDA
   SDA (StatusRecord) Status
   TYPE (fe) FeModel
   TYPE (SurfForces) force

   CALL Surf%GetFeModel(FeModel)
   DO WHILE (.TRUE.)
      CALL Grid%SurfaceForces(force)
      load = interp(force, FeModel)
      solve(load, FeModel, deflect)
      IF ( deflect .GT. tol ) THEN
         CALL Status%SetDone
         CALL Surf%GetFeModel(FeModel)
      ELSE
         CALL Surf%PutDeflected(deflect)
      ENDIF
   END DO

END FeSolve
```

Figure 6: Finite Element Solver

```
SDA TYPE StatusRecord
   REAL drag
   LOGICAL Done = .FALSE.
   PRIVATE ConvError, Done

CONTAINS
   SUBROUTINE GetDone WHEN Done
      Done = .FALSE.
   END GetDone

   SUBROUTINE SetDone WHEN .NOT. Done
      Done = .TRUE.
   END SetDone

END StatusRecord
```

Figure 7: SDA for Status