Opus: A Coordination Language for Multidisciplinary Applications*

Barbara Chapman[1] **Matthew Haines**[2] **Piyush Mehrotra**[3]
Hans Zima[1] **John Van Rosendale**[3]

[1]Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1090, Vienna, Austria
{barbara,zima}@par.univie.ac.at

[2]Computer Science Department
University of Wyoming
Laramie, WY 82071-3682
haines@cs.uwyo.edu

[3]Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Mail Stop 132C
Hampton, VA 23681-0001
{pm,jvr}@icase.edu

Abstract

Data parallel languages, such as High Performance Fortran, can be successfully applied to a wide range of numerical applications. However, many advanced scientific and engineering applications are multidisciplinary and heterogeneous in nature, and thus do not fit well into the data parallel paradigm. In this paper we present Opus, a language designed to fill this gap. The central concept of Opus is a mechanism called ShareD Abstractions (SDA). An SDA can be used as a computation server, i.e., a locus of computational activity, or as a data repository for sharing data between asynchronous tasks. SDAs can be internally data parallel, providing support for the integration of data and task parallelism as well as nested task parallelism. They can thus be used to express multidisciplinary applications in a natural and efficient way. In this paper we describe the features of the language through a series of examples and give an overview of the runtime support required to implement these concepts in parallel and distributed environments.

1 Introduction

With the arrival of teraflop architectures, the complexity of simulations being tackled by scientists and engineers is increasing exponentially. Many of these simulations are of a complex, "multidisciplinary" nature, constructed by pasting together modules from a variety of related scientific disciplines. This raises a host of

*This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NASA-19480, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

new software integration issues. While data parallel languages, like HPF [21], are well suited to exploiting the parallelism in each module [10], they offer little support for integration and also do not exploit the coarse grained parallelism that multidisciplinary applications frequently provide.

One example of a multidisciplinary application is environmental simulation. One might, for example, have a sequence of models, such as a) a swamp biology model for the Everglades, b) a hydrothermal model for the Gulf stream, c) a mesoscale climate model and d) a solar radiation model. The goal is then to interconnect these models into a multidisciplinary model subsuming the original models together with their various couplings.

Another example is multidisciplinary optimization (MDO). Designing a modern aircraft, for example, requires a wide variety of interacting disciplines: aerodynamics, propulsion, structural analysis, 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 set of complex scientific and engineering programs from different disciplines, into a coherent whole capable of effective multidisciplinary optimization.

Implementing multidisciplinary applications raises a number of complex programming issues. One is that the constituent programs being glued together are typically written by different groups, using different data structures and approaches. Moreover, the mix of programs involved typically changes over time. In the environmental simulation, for example, one might find it necessary to add a model of airborne particle transport to correctly predict solar heating. Similarly, in MDO of an aircraft, one might need to replace a simple linear flow solver by a more sophisticated Euler or Navier Stokes code.

In such large scale programming projects, statically forming a "task graph" and coupling tasks via "message plumbing" is virtually unworkable. A much more flexible software environment appears to be critical. At the same time, one wants to effectively exploit the parallelism both within and across the separate discipline models. Exploiting the coarse-grained parallelism in multidisciplinary applications requires facilities for spawning and synchronizing collections of tasks, each of which might contain internal data-parallelism.

We have recently designed a coordination language, called **Opus**, targeted towards such applications. It provides a software layer on top of data parallel languages, such as HPF, designed to address both the "programming in the large" issues and the parallel performance issues arising in complex multidisciplinary applications.

The heart of Opus is a new mechanism, called **S**hare**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. Different SDAs represent distinct address spaces, hence Opus tasks do not directly share data. Instead, interaction between tasks is accomplished by invoking methods in other SDAs. Thus, a set of tasks may share a pool of common data by creating an SDA of the appropriate type and making the data 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.

The runtime system supporting Opus utilizes lightweight, user-level threads that are capable of supporting both intra- and inter-processor communication primitives in the form of shared memory, message-passing, and remote service requests [20]. This allows the independently executing SDA tasks to freely share the underlying parallel resources.

The remainder of the paper is organized as follows: The next section discusses the language extensions defined in Opus and their use. Section 3 presents a couple of multidisciplinary applications, using the concepts introduced in Section 2. Section 4 outlines the runtime support necessary for implementing these extensions. This is followed by a section on related work and a brief set of conclusions.

2 The Opus Language

There are a number of constraints which must be satisfied by any general framework which supports the coupling of multiple programs into complex multidisciplinary codes. In particular, we have identified the following requirements:

- The separate programs should be "encapsulated" into modules in a way that respects their separate name spaces.
- Coupling between modules should be at the highest level (as opposed to having message-passing constructs throughout the code).
- Both task-level parallelism between modules, and data parallelism within each module should be expressible.
- Flexible and general synchronization mechanisms should be provided to allow the programmer maximal freedom in exploitation of task-level parallelism.

The first two of these requirements are motivated by software-engineering considerations. Their purpose is to simplify the combination of component modules, enable the definition of clear interfaces between modules, and allow modules to be intermixed without rewriting their internal code. This is in contrast to message-passing models, which combine modules with no clear interface definition.

The other two requirements are needed for performance. Multidisciplinary codes are among the largest and most computationally intensive codes, so that any language designed for such applications must have the potential to fully exploit highly parallel architectures.

To fulfill these requirements, Opus introduces a new construct called a **ShareD Abstraction (SDA)**. This concept supports the development of MDO codes by providing data and method encapsulation. SDAs can be used as computation servers as well as shared data repositories. We use the well-developed HPF facilities for data parallelism within each SDA, while borrowing ideas from operating systems for inter-module communication and task-level parallelism.

In this section, we describe the most important constructs of Opus and illustrate them by applying them to the standard *producer-consumer problem.* A simple meteorological coordination problem and a more challenging example – taken from the domain of aircraft design – will be discussed in the next section.

2.1 The 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.

We summarize the way in which these features are used to build an Opus application below. A full description of the language features can be found in [24]. 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.

Each SDA 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.

2.2 The Producer-Consumer Problem

We introduce the syntax and semantics of the Opus language by developing an Opus solution to the standard producer-consumer problem. This simple problem, in which a set of producers generate data which are processed by a set of consumers, is also the basis for a number of real-world applications. Our version creates a system in which each individual producer and consumer operates independently. Synchronization between them is provided by controlling their access to a bounded FIFO buffer.

To do this, the first step is to define an SDA type which encapsulates the data structures required to implement the bounded buffer along with the access methods which permit producers to write to the buffer and consumers to read from it.

```
SDA TYPE buffer_type(size)
   INTEGER :: size
   REAL :: fifo(0:size-1)
   INTEGER, READ_ONLY :: count = 0        ! number of full elements in fifo
   INTEGER :: px=0          ! producer index
   INTEGER :: cx=0          ! consumer index
    ...
CONTAINS
    ! method part
END buffer_type
```

The above fragment shows the data structure created to define a buffer which may hold up to *size* data items of type REAL. Specification of the value of *size* is deferred until the actual creation of an SDA (see below). The variable *count* keeps track of the current number of elements in the buffer, while *px* and *cx* point to the current index positions for producers and consumers respectively.

In contrast to Fortran modules, the internal variables of an SDA type are by default private, i.e., are accessible only from the methods associated with the SDA. The keyword **PUBLIC** can be used to change this default for the whole SDA or to control the accessibility of individual variables. Opus extends Fortran by supporting the attribute **READ_ONLY**, which allows SDA variables, such as *count* above, to be accessed but not modified from outside.

Next, access methods for reading from and writing to the buffer have to be defined. The producers may write data to the buffer only if the buffer is not full, while consumers may read data only if the buffer is not empty. Opus enables conditional execution of a method by permitting a *condition clause*, containing a side-effect free logical expression, to be associated with a method. 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 can happen as a result of another method call that changes variables on which the condition depends.

Our formulation defines two methods: subroutines *get* and *put* for reading from and writing to the buffer respectively. These are shown below:

```
 SUBROUTINE put(x) WHEN (count .LT. size)  ! the condition clause expresses the assertion
                                           ! "buffer is not full"
   REAL, INTENT(IN) :: x
   fifo(px) = x                            ! Put x into first empty buffer element
   px = MOD(px+1,size)
   count = count + 1
 END

 SUBROUTINE get(x) WHEN (count .GT. 0)     ! the condition clause expresses the assertion
                                           ! "buffer is not empty"
   REAL, INTENT(OUT) :: x
   x = fifo(cx)                            ! Read next full buffer element
   cx = MOD(cx+1,size)
   count = count - 1
 END
```

The condition clauses control access to the buffer, allowing *put* methods to be executed only when the buffer is not full and *get* methods to executed only when the buffer is not empty. If we combine these methods with the data declarations defined above, the interface between the producer and consumer tasks is fully specified.

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, e.g., *fifo* and *count*, without interference from other activations of the methods.

The dummy arguments of an SDA type specification are all of intent IN and therefore passed in by value. Methods are arbitrary procedures, and may have arguments of any intent, which are passed with copy-in/copy-out semantics.

The producer and consumer tasks must now be asynchronously activated and linked with the SDA in such a way that they are able to write and read the buffer, respectively. This is implemented as follows. First, an SDA variable, *buffer*, of the SDA type *buffer_type* is declared as shown below:

```
INTEGER  buffersize
SDA(buffer_type)  buffer

READ  *, buffersize
CALL buffer%CREATE(buffersize)
```

CREATE is an implicit method which is called to create the SDA object to be associated with the variable *buffer*. The variable *buffersize* is passed in as the actual argument which is associated with the formal argument *size* and is used to allocate the internal data structures of the SDA. *CREATE* allocates and initializes the SDA object. The user may augment the system initialization by defining an *INIT* method which is implicitly called after the call to *CREATE*. Opus provides other methods which are implicitly declared for all SDA types: *SAVE*, *LOAD*, and *TERMINATE*.

SAVE permits the saving of the internal state of an SDA to a named external object, while *LOAD* allows the creation of an SDA object based on an external object. *SAVE* and *LOAD* provide the minimum language support required for dealing with persistent SDAs. For convenient use of this mechanism in real applications several extensions are desirable. We are currently studying additional language features focusing on *partial saving*, the relaxation of the type conformity requirements in LOAD, and input/output, in particular using *smart files* [18] for external storage of the data.

In general, the lifetime of an SDA object extends from the time it is created to the time that the execution leaves the scoping unit in which the SDA declaration was originally processed. At this time the SDA is implicitly terminated. The *TERMINATE* method can be called to explicitly terminate an SDA and free its associated storage.

Note also that the language provides facilities to specify system resources at the time of initialization of the object either through the *CREATE* or *LOAD* methods (see next section for some examples).

Once the SDA object has been created, its public data can be accessed and the associated methods called using a syntax similar to that used for derived types in Fortran. Thus, for example, the consumers can invoke the *get* method for the SDA *buffer* as follows to access the next data element.

```
CALL buffer%get(A)
```

The above statement designates a **synchronous method activation** which will block the caller until the method call returns.

In order to support concurrent activity, Opus also provides **asynchronous method activation** in which the caller is not blocked by the method call. For example, in the code below, a *spawn* statement is used to invoke the method *get* asynchronously.

```
EVENT E
 ...
E = SPAWN buffer%get(A)
 ! Do other work.
WAIT(E)
```

The *spawn* statement returns an *event* which is assigned to the *event variable E*. The calling unit can continue its computation and use the event variable in a *wait* statement, as shown above, to wait for the completion of the associated method call. This allows the caller and the invoked method to execute in parallel, in this case overlapping computation with "getting" data elements from the buffer.

A nonblocking alternative to the wait statement, **TEST** (E), allows the caller to test for the completion of an asynchronous method invocation. It returns the current completion state.

As with SDA methods, the *spawn* statement can also be used with generic Fortran subroutines to generate concurrent activity. Thus, in the full producer-consumer code, as shown in Figure 1, *np* copies of the subroutine *produce* and *nc* copies of the subroutine *consume* are spawned as asynchronously executing tasks. Each is passed the SDA variable *buffer* which they use as a shared resource for communicating values. Note that we have omitted the code for terminating these tasks.

3 Multidisciplinary Applications Using Opus

Multidisciplinary applications, including the important subclass of multidisciplinary optimization (MDO) problems, are commonly formed by combining data parallel units from various disciplines to create a single application. With the increase in the size of computing systems available and the improved access to them, development of such applications, and the complexity of the coupling between the individual components is steadily increasing. Below we introduce two examples. The first of these, taken from meteorology, has a simple and well-defined interaction between its two component modules. The next example is a simplification of an MDO application for aircraft design with rather more complex interaction patterns.

3.1 Opus for Data Parallel Applications

One situation in which the kind of interaction described in the producer-consumer program might occur in practice is the coupling of a global numerical weather prediction (NWP) model with a limited area forecast model. In this case, the boundary areas of the limited area model are refreshed by the interpolation of results

```
PROGRAM Consumer_Producer
   INTEGER   np, nc, buffersize
   SDA(buffer_type)  buffer

   READ(np,nc,buffersize)
   CALL buffer%CREATE(buffersize)

   DO i= 1, np                          !Spawn producers
     SPAWN produce(buffer, ...)
   END DO
   DO i= 1, nc                          !Spawn consumers
     SPAWN consume(buffer, ...)
   END DO
      ...
END
```

```
SDA TYPE  buffer_type(size)
   INTEGER          :: size
   REAL    :: fifo(0:size-1)
   INTEGER, READ_ONLY ::  count=0
   INTEGER ::  px=0, cx=0
 CONTAINS
   SUBROUTINE put(x) WHEN (count .LT. size)
     REAL, INTENT(IN) :: x
     fifo(px) = x; px = MOD(px+1,size); count = count + 1
   END

   SUBROUTINE get(x) WHEN (count .GT. 0)
     REAL, INTENT(OUT) :: x
     x = fifo(cx); cx = MOD(cx+1,size); count = count - 1
   END
END buffer_type
```

```
SUBROUTINE produce(b, ...)
   SDA(buffer_type) b
    ...
   DO WHILE   (.TRUE.)
     ! produce a data item A
     CALL b%put(A)
   END DO
END produce
```

```
SUBROUTINE consume(b, ...)
   SDA(buffer_type) b
    ...
   DO WHILE  (.TRUE.)
     CALL b%get(A)
     ! consume A
   END DO
END consume
```

Figure 1: Producer/Consumer Problem Using Opus

from the global model at time steps corresponding to fixed intervals over the time period of the prediction. We use this very simple coupling example to consider the data parallel requirements of an Opus application.

We assume that the global NWP program *global* and the local NWP program *local* have been independently developed and that they are available as distinct HPF applications. A simple data interface is required for their coupling.

The program *global* will write the data set corresponding to the boundary areas of the limited area model to an SDA at the appropriate intervals, from which it will be read in by *local*. In order to maintain accuracy in the limited area computation, it is important that *local* receives the data sets from *global* in their chronological order and that all of them be processed. The amount of data being transferred dictates that only a small number of data sets be stored at any time; here, we assume that only one such data set is to be saved in the SDA for reading by *local*.

The following code fragment shows part of the definition of the SDA type *shared_metdata* which is used with a series of methods to read and write a number of different fields of meteorological data. We show just a few variables here: in practice, there are likely to be on the order of half a dozen different quantities. HPF directives are used to distribute the arrays by blocks to the processors on which the SDA is executed.

```
      SDA TYPE shared_metdata(size)
!HPF$ PROCESSORS P(number_of_processors())       ! HPF directive to specify the processor set
        INTEGER :: size
         ! data fields used to save boundary values:
        REAL    :: temp(size)
        REAL    :: xvelo (size)
          ...
!HPF$ DISTRIBUTE (BLOCK) ONTO P:: temp, xvelo    ! HPF directive to distribute meteorological
                                                 ! data by blocks across the processors
        LOGICAL :: tempmarker  = .FALSE.  ! variable used to indicate  whether unread data is
                                          ! stored in the SDA
          ...
       CONTAINS
         SUBROUTINE puttemp(restemp) WHEN (tempmarker .EQ. .FALSE.)
                   ! puttemp stores global temperatures in the SDA array temp
         REAL, INTENT(IN)    :: restemp(size)
!HPF$    DISTRIBUTE (BLOCK) :: restemp
           temp = restemp
           tempmarker = .TRUE.
         END
          ...
         SUBROUTINE gettemp(boundtemp) WHEN (tempmarker .EQ. .TRUE.)
                   ! gettemp reads global temperatures from the SDA array temp
         REAL, INTENT(OUT)     :: boundtemp(size)
!HPF$    DISTRIBUTE (BLOCK) :: boundtemp
           boundtemp = temp
           tempmarker = .FALSE.
         END
       END shared_metdata
```

The next step is to create an SDA of the above type and spawn the local and global codes which would use the SDA to transfer data. This is shown in the code fragment below:

```
!HPF$ PROCESSORS R(32)
     SDA(shared_metdata)  boundary
```

```
    ...
CALL boundary%CREATE(insize) ON (PROCESSORS R(1:16) )

SPAWN global(boundary, ...) ON (PROCESSORS R(17:32) )
SPAWN local(boundary, ...)  ON (PROCESSORS R(1:16) )
    ...
```

In this coordination application, the two methods are asynchronously invoked on two distinct sets of processors of the available computing system to run the weather codes (these may well be on different computers in practice). An HPF directive has been used to declare the processors involved; it specifies both the number of processors and gives them a global name. This is then referred to in the method calls which create the SDA and asynchronously spawn the global and local codes. Thus the user can ensure that the two applications run on different sets of processors and that an appropriate set of processors is allocated for each code. In the above code, a decision has been made to locate the data produced by *global* on the same processors as the code, *local*, which will read them. HPF notation has also been used to distribute the data associated with the SDA. We may assume that the specification of this distribution enables the reading of data to be performed locally when the method *gettemp* is invoked.

In practice, a non-trivial filter will be required to transfer data between two such models: not only will the grid points have different distances, the models may well use different coordinate systems. We do not consider this aspect here.

3.2 MDO for Aircraft Design

In this subsection we present a short description of the multidisciplinary design of an aircraft and then discuss how one version could be encoded using the Opus language constructs. The overall goal of the 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.

The analysis phase consists of the various discipline codes, such as aerodynamic analysis, structural analysis, controls, etc., interacting with each other to analyze the current definition of the aircraft. Some disciplines, such as aerodynamic or structural analysis, exhibit a large degree of internal parallelism and thus require substantial physical resources for execution. However, other disciplines are generally simpler and should most likely be executed sequentially. The amount of data exchanged during the analysis phase is dependent on the disciplines involved and ranges from a few bytes to millions of bytes. Sometimes, this data needs to be "massaged", or filtered, before it can be used. For example, pressures produced at the aerodynamic grid points by the flow analysis code have to be integrated to produce forces at the structural grid points for structural analysis.

The interactions between the discipline codes can take different forms depending on the problem at hand and the target environment. In a sequential environment, the various discipline codes are generally executed as a pipeline. In a simple parallel variant, multiple versions of the analysis pipeline can be executed on slightly perturbed values of the design variables in order to obtain the required derivatives using finite differences. In more complex parallel versions, such as the one we describe here, the discipline codes execute asynchronously, with data being exchanged at various points in the code, such as at the boundaries of the internal optimization cycles. For this latter approach, the data exchanges must be synchronized to ensure that consistency is maintained.

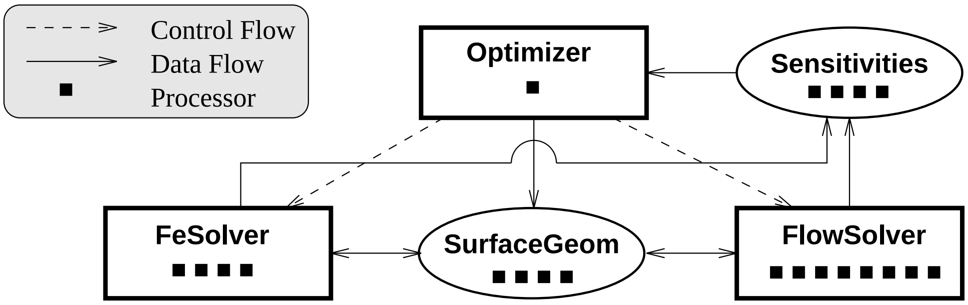


Figure 2: Data flow in a simple MDO application for aircraft design

3.2.1 Opus Code

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 a full aircraft configuration would require a number of other discipline codes, such as controls, performance analysis, propulsion, etc., 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 representing computational activities are represented by rectangles and the SDAs representing data repositories are represented by ovals. The *Optimizer* is the main task and coordinates the execution of the entire MDO application.

As shown in Figure 3, the *Optimizer* creates the following SDAs: the data repositories *SurfaceGeom* for sharing geometry and flow data between the two computational tasks, and *Sensitivities* for storing the sensitivity derivatives, and the computational tasks *FeSolver* for structural analysis of the aircraft configuration, and *FlowSolver* for aerodynamic analysis. Since the tasks *FeSolver* and *FlowSolver* use the other two SDAs to transfer data, the latter are passed in as arguments as the former are being created. The *on clauses* associated with the create statements specify the resources to be used for the SDAs as shown in the code fragment from Figure 3 reproduced below:

! – 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**(...) **ON** (**MACHINE**="XYZ", **PROCESSORS** = 4)
CALL FlowSolver%**CREATE**(...) **ON** (**MACHINE**="XYZ", **PROCESSORS** = 8)

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 machine "XYZ" 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 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 4) will be instantiated as an array of four processors. That is, for the SDA instance *SurfaceGeom*, the HPF function *number_of_processors()* will return four. As indicated before, the data within the SDA can now be distributed using the full power of the HPF mapping directives.

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 4, stores the geometry in the variable *base*, initializes the variable *deflected*,

```
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="XYZ", PROCESSORS=8)
 ! – initialize  geometry
  geom = GenBaseGeom(...)

 ! – optimization loop
  converged = .FALSE.
  DO WHILE (.NOT converged)
     SPAWN 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 3: Main program: Optimizer

```
    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 (DeflectFull .EQ. .FALSE.)
        TYPE(surface), INTENT(IN) :: d
        deflected   = d
        DeflectFull = .TRUE.
      END

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

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

Figure 4: 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 FeSolve
```

Figure 5: Finite Element Solver

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 5, 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 Opus Runtime Support

In the previous two sections we have presented features of Opus and examples showing how these features can be used to encode interacting asynchronous data parallel tasks. In this section we describe the runtime system required to support these features.

The Opus runtime system consist of two layers (see Figure 6):

- a language-specific layer, providing the functionality for managing SDAs and their interaction via method calls, and
- a language-independent layer, which provides support for thread-based data parallelism in parallel distributed environments.

We discuss first the thread-based layer and then describe the implementation of method invocation, including the handling of distributed arguments in the Opus runtime system.

4.1 Lightweight Threads

As described in the previous sections, SDAs can be configured either as computation servers or as data servers. In general, the computation server tasks and the data servers will utilize the same (or overlapping) physical resources. Thus, any given processor in the system might be responsible for the simultaneous execution of multiple, independent SDAs. Execution of these SDAs could be implemented on Unix-based systems by mapping each unit to a process. However, this process-based approach has several drawbacks, including the inability to control scheduling decisions for the SDA methods, the inability to share addressing spaces between SDAs, and costly context switching between SDAs. In light of these disadvantages, our runtime system utilizes lightweight, user-level threads to represent the parallelism within and among SDAs. This decision is consistent with most other runtime systems supporting parallel or concurrent programming languages [7, 14, 4].

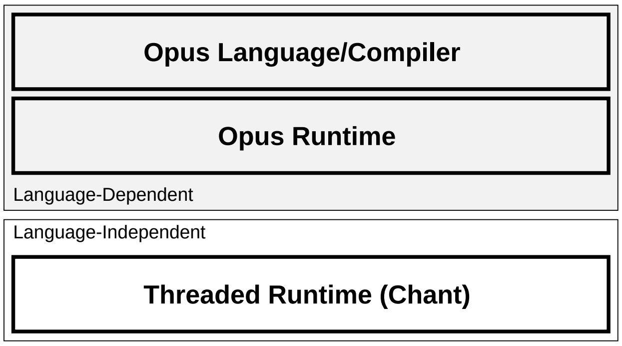


Figure 6: Runtime layers for SDA support

A lightweight, user-level thread is a unit of computation with minimal context that executes within the domain of a kernel-level entity, such as a Unix process or Mach kernel thread. Lightweight threads are becoming increasingly useful in supporting language implementations for both parallel and sequential machines by providing a level of concurrency within a kernel-level process.

The language-independent layer of the OPUS runtime system is based on Chant. Chant provides both a standardized interface for thread operations (as specified by the POSIX thread standard [25]) and communication among threads using either point-to-point primitives (such as those defined in the MPI standard [23]) or remote service requests. Chant also supports data parallel groups of threads (called ropes) for executing collective operations, such as broadcast and reductions. A description of Chant, and its current status, can be found in [17, 19].

The Opus runtime system is primarily concerned with the management of SDAs and their interaction via method calls. The underlying HPF runtime system will deal with issues of data parallelism and distribution. In the initial design, we concentrate on the interaction of SDAs through method calls (namely method invocation and argument handling), and have taken a simplified approach to resource management. We presume that all the required resources are statically allocated and the appropriate code is invoked where necessary. We will later extend the design of the runtime system to support dynamic acquisition of new resources.

The interaction between SDAs requires runtime support for both method invocation and method argument handling. We now explore these issues in further detail.

4.2 SDA Method Invocation

The semantics of SDAs places two restrictions on method invocation:

- each method invocation has *exclusive* access to the SDA data (i.e., only one method for a given SDA can be active at any one time), and
- execution of each method is guarded by a condition clause, which must evaluate to *true* before the method code can be executed.

An SDA method call can be either synchronous or asynchronous. A synchronous method call will suspend the calling program until the SDA method returns; an asynchronous method invocation will allow the caller to continue execution and test for method termination with an event variable.

We can view an SDA as being comprised of two components: a *control structure* which executes the SDA methods in accordance with the stated restrictions, and a set of SDA *data structures*. To enable proper execution of SDAs, each SDA method is compiled into three functions:

1. The *method code*. This function embodies the method code as specified by the programmer. It uses a generic method call interface that permits the invocation of all SDA method calls in a uniform manner.

```
Do forever {
   Wait for method request for method m from caller
   Create new execution record X
   Send X to caller as acknowledgment
   Post receives for input arguments from caller
   Enqueue X in queue for method m
   Repeat
      Select next ready method request Y
      Execute method Y
      Send results to caller
      Send completion signal to caller
      Dequeue Y
   Until no more method requests are ready
}
```

Figure 7: Pseudocode for an SDA main loop

2. The *condition function*. This is a boolean function that evaluates the condition clause that may be associated with an SDA method. The condition clause must be locally evaluated to ensure that race conditions do not occur.

3. The *method interface*. This is a stub function that provides the method's public interface to the calling units and is used to access the SDA method code from another program unit.

Since all SDAs are servers, either for data or computation, each instance of an SDA is represented by a server loop (as depicted in Figure 7) which waits for messages from the method interfaces of other units and takes appropriate action as specified by the message. The SDA instance incorporates a data structure that includes pointers to the *condition* and *method* functions for each method along with a queue of outstanding method invocation requests.

The algorithm in Figure 7 depicts the main loop of an SDA server. On receiving a message from a *method interface* routine, the SDA creates a new execution record including a unique identification for the request. This record is sent back to the caller as acknowledgment. The SDA gathers any input arguments using non-blocking receives (so as not to impose an artificial ordering on the incoming messages) and enqueues the execution record in the appropriate list. The SDA then selects the next method request which is ready for execution. A method request is ready for execution if all its arguments have been received and the associated condition is *true*. After execution of this method request, the results, if any, are sent back to the caller. A completion signal is also sent back to the caller and the execution record is dequeued from the method request list. This reevaluation of condition functions is repeated until no further methods can be executed, at which time the SDA continues waiting for further method requests.

Figure 8 shows a generic method interface routine used by the calling task to invoke a method. After the method request is sent, the caller waits for an acknowledgment and then sends the values of the input arguments to the callee. If the method activation was synchronous, the caller waits for the results and for the completion signal before returning. If the method activation was asynchronous, it posts non-blocking receives for the results and the completion signal. The execution record is returned to be stored as the event associated with the method activation. This allows the caller to continue execution without the completion of the method call. The event (i.e., the execution record) can be used later in a wait or test statement to test for the completion of the method call.

```
Send method request to SDA
Wait for execution record X as acknowledgment
Send actual arguments to callee
If activation_type = asynchronous
    Post receives for results
    Post receive for completion signal
    Return X
else
    Wait for results
    Wait for completion signal from callee
endif
```

Figure 8: Pseudocode for a method call interface

4.3 Distributed Argument Handling

In the previous subsection, we described the protocol for invoking methods under the implicit assumption that both the calling SDA and the called SDA run on a single processor. However, the language allows both to be distributed; furthermore, the distributions of the actual and the formal arguments of method calls may not match. Thus, the Opus runtime system must have a mechanism for redistributing data at method invocation time. To examine the details of our prototype implementation, let us consider what happens when a distributed task calls a method in a distributed SDA, referring to the pictorial representation in Figure 9.

If an SDA type is internally distributed, an SDA instance of this type is represented by a *rope*, which is a data parallel group of threads spread across the set of processors. One of the threads is designated the *leader* thread while the other threads are *worker* threads. Method invocation between distributed SDAs then works as follows (the pseudocode for the main loop of the SDA leader and the workers of a distributed SDA is shown in Figure 10):

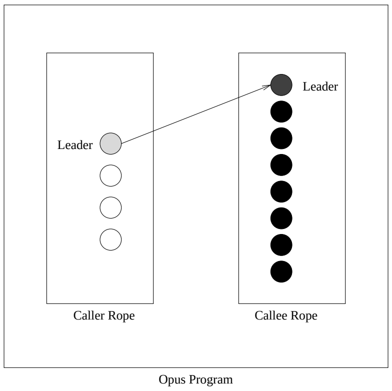
1. The leader thread of the rope associated with the caller (the *caller rope*) sends a method request message to the leader thread of the rope associated with the called SDA (the *callee rope*) (Figure 9.1). Along with other information, this message also contains the distribution specifications for the actual method arguments.

2. The leader of the callee rope then creates an execution record containing the distribution specifications of the dummy method arguments and sends it back to the leader of the caller rope. It also notifies its workers of the method request (Figure 9.2), along with the distribution specifications of the actual arguments.

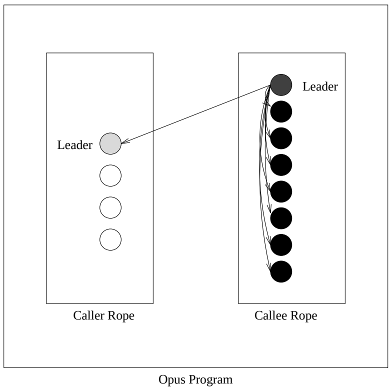
3. The leader of the caller rope then informs all its workers of the dummy argument distribution information it has received. At this point, all threads involved in the method invocation have the distributions of both the dummy and actual arguments, and can create their own communication schedules as discussed below (Figure 9.3).

4. Once the communication schedules have been computed, the threads of the caller rope send data messages directly to the appropriate threads of the callee rope (Figure 9.4). The data is received by these threads through non-blocking receives.

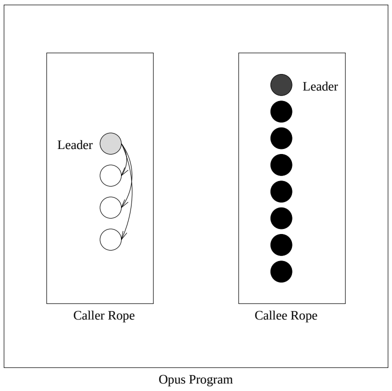
5. The leader of an SDA rope chooses the next ready method to execute and informs all its workers. The method is executed and all threads of the callee rope send any return messages back to the threads of



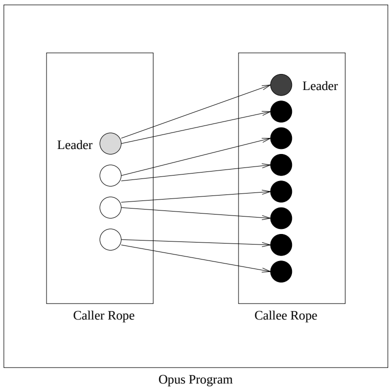
1. Caller leader thread sends method request to callee leader thread with actual argument distributions.



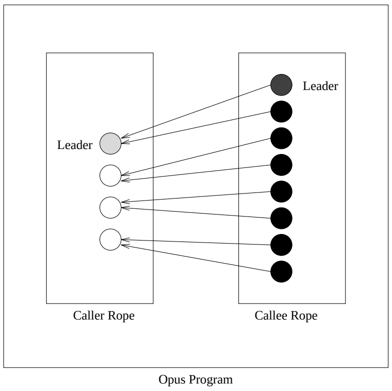
2. Callee leader thread notifies its workers and ACKs the request with dummy argument distributions.



3. Caller leader sends callee distribution information to all its workers. All threads in the caller and callee ropes compute communication schedules.



4. Caller threads send data messages to appropriate callee threads directly.



5. When method execution has finished, the callee threads send any return messages to the caller threads. This completes the method call.

Figure 9: Illustration of the method invocation process for distributed SDAs

```
SDA leader:
   Do forever {
      Wait for method request for method m from caller
              including distributions of actual arguments
      Create new execution record X
              including distributions of formal arguments
      Send X to leader thread of caller as acknowledgment
      Send X to all workers
      Compute communication schedule
      Post receives for input arguments from caller
      Enqueue X in queue for method m
      Repeat
         Select next ready execution record Y
         Send Y to all workers
         Execute method Y
         Send results to caller
         Send completion signal to leader thread of caller
         Dequeue Y
      Until no more method requests are ready
   }
```

```
SDA Workers:
   Do forever {
      Wait for message from leader
      If new method execution record X received
         Compute communication schedule
         Post receives for input arguments from caller
         Enqueue X in queue for method m
      else
         Execute method X
         Send results to caller
         Dequeue X
      endif
    }
```

Figure 10: Main loops for leader and workers in a distributed SDA

the caller rope using the previously computed communication schedule (Figure 9.5). The leader of the callee rope then sends a completion signal to the leader of the caller rope.

The leader of the callee rope controls which method request is to be executed next, and thus sends to its worker threads messages for new method requests or for execution of already queued requests. In the former case, as shown in Figure 10, the worker threads independtly compute their communication schedules and post their receives. In the latter case, they execute the method and send back the results. We currently assume that the condition code is executed solely by the leader and only uses information which is replicated across the rope and thus can be accesses locally by the leader. local to the leader.

Determining the communication schedule, i.e., what elements of an array are to be sent or received from which thread, is a complex task. Several groups have been studying algorithms and heuristics to determine the most efficient schedule [2, 11, 26, 16, 22, 27, 31]. We have adopted (and augmented) the finite state machine (FSM) method for local address set calculation developed by Chatterjee *et al.* [11] in our current prototype. The FSM method exploits the repeating patterns of local array indices to determine the elements of a distributed array that each thread owns. Since all threads can do this calculation simultaneously, there is no gather/scatter operation required. We have extended this work by creating a second FSM such that, for each local element of the array yielded by the original FSM, the thread can determine the destination thread it must communicate with. Each thread in the sender creates a list of elements for each destination thread which is then aggregated into a single message for each other thread and transmitted. Thus, each destination thread will receive at most one message from each sending thread. In addition, each receiving thread can use the same FSM method along with the sender's distribution information to determine from whom it should receive messages and what the contents will contain. Consequently, the messages contain only raw data, eliminating the overhead of transmitting indices.

We have developed a prototype implementation of the Opus runtime system, which is currently running on a cluster of workstations using p4 and the Intel Paragon using NX. This implementation handles distributed arguments in synchronous method calls. A complete description of the system and some preliminary results can be found in [20].

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 [3]. 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 [13] 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 interact 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. 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 processes 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.

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

Recent work at Argonne and Syracuse [15] integrates HPF with the message passing standard MPI. In this approach, data parallel HPF tasks may exchange distributed data structures by directly using calls to MPI communication functions.

The **Fx** Fortran language extensions developed at CMU [29, 28] include *parallel sections* that allow the concurrent activation of subroutines as *tasks*. Tasks communicate by through 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 [1] 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.

Orca [5] provides an object model similar to SDAs called *abstract data types* (ADTs). Both ADTs and SDAs represent abstract data types that can be distributed over a set of processors using conventional data parallel mapping directives. Both apply operations to their elements using the owner-computes rule. Aside from implementation issues, the main difference between ADTs and SDAs is in the "server" nature of the SDA. All SDAs run implicit server loops to handle incoming requests, and SDA methods can be invoked both synchronously and asynchronously, where the decision can be made at the call site. This allows SDAs to behave as computation servers as well as data servers. Orca objects deliberately lack such a server, to allow concurrent read operations on different copies of an object.

SVM Fortran [6] 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 PVM [30], CC++ [8] and Strand [12]. Most of these approaches do not address the issue of integrating task and data parallelism.

6 Conclusions and Future Research

Complex scientific applications, such as multidisciplinary optimization, provide opportunities for exploiting multiple levels of parallelism, but also raise complex programming issues. The coordination language Opus, presented in this paper, supports the multiple levels of parallelism arising in multidisciplinary applications, and also provides support for software engineering issues arising when integrating codes from individual disciplines into a single working application.

A partial implementation of Opus has been built, using the Chant runtime system. Performance of a simplified multidisciplinary application code has been studied using this implementation. The cost of a typical SDA method call with distributed arguments appears to be reasonable and our design scales with the number of processors. Given these preliminary results, a full prototype implementation of Opus has begun. Since Chant runs on a large number of multiprocessor platforms, this prototype will be widely portable, and should prove useful in a number of important applications. We also plan to explore the research issues of supporting parallel method calls within the same SDA and condition evaluation based on distributed data structures.

References

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

[2] A. Ancourt, F. Coehlo, F. Irigoin, and R. Keyrell. A linear algebra framework for static HPF code distribution. In *Proc. of the 4th Workshop Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

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

[4] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings, 1993.

[5] 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.

[6] R. Berrendorf, M. Gerndt, W. Nagel, and J. Prummer. SVM FORTRAN.. Technical Report KFA-ZAM-IB-9322, Research Center Jülich(KFA), Germany, November 1993.

[7] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek. Panda: A Portable Platform to Support Parallel Programming Languages. *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, 213–226, September 1993.

[8] 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.

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

[10] B. Chapman, P. Mehrotra, and H. Zima. Extending HPF for Advanced Data Parallel Applications. *IEEE Parallel and Distributed Computing*, 2(3):59–70, Fall 1994.

[11] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. In *Symposium on Principles and Practice of Parallel Programming*, pages 149–158, May 1993.

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

[13] I.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.

[14] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems. Technical Report, Argonne National Labs, 1993.

[15] I. Foster, D. R. Kohr, Jr., R. Krishnaiyer, and A. Choudhary. Double Standards: Bringing task Parallelism to HPF via the Message Passing Interface. Proc. Supercomputing 96, Pittsburgh, PA. (to appear).

[16] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed memory machines. Technical Report OSU-CISRC-4/94-TR19, The Ohio State University, Department of Computer and Information Science, Columbus, OH 43210, March 1994.

[17] M. Haines, P. Mehrotra, and D. Cronk. Ropes: Support for collective operations among distributed threads. ICASE Report 95-36, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681, May 1995 (to appear in Scientific Programming).

[18] M. Haines, P. Mehrotra, and J. Van Rosendale. SmartFiles: An OO approach to data file interoperability. ICASE Report 95-56, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681, July 1995. To appear in OOPSLA 95.

[19] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing 94*, pages 350–359, Washington, D.C., November 1994. Also appears as ICASE Technical Report 94-25.

[20] M. Haines, B. Hess, P. Mehrotra, J. Van Rosendale, and H. Zima. Runtime support for data parallel tasks. In *Frontiers 95*, February 1995.

[21] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Version 1.1, November 1994.

[22] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *Proceedings of the International Conference on Supercomputing*, pages 180–184, Barcelona, Spain, July 1995. ACM Press.

[23] Message Passing Interface Forum. *Document for a Standard Message Passing Interface*, Version 1.1, June 1995.

[24] *Opus Reference Manual*, ICASE Internal Report (under preparation).

[25] IEEE. *Threads Extension for Portable Operating Systems (Draft 7)*, February 1992.

[26] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. Technical Report UILU-ENG-94-2213, CRHC-94-09, University of Illinois, April 1994.

[27] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[28] 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, 1994.

[29] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, San Diego, CA, May 1993.

[30] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[31] A. Thirumalai and J. Ramanujam. HPF Array statements: Communication generation and optimization. In *Proc. 3rd Workshop on Language Compilers, an Runtime Systems for Scalable Computers*, Troy, NY, May 1995.