Opus: Heterogeneous Computing With Data Parallel Tasks*

Erwin Laure[a] *Piyush Mehrotra*[b] *Hans Zima*[a]

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

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

May 5, 1999

Abstract

The coordination language Opus is an object-based extension of High Performance Fortran (HPF) that supports the integration of coarse-grain task parallelism with HPF-style data parallelism. In this paper we discuss Opus in the context of multidisciplinary applications (MDAs) which execute in a heterogeneous environment. After outlining the major properties of such applications and a number of different approaches towards providing language and tool support for MDAs we describe the salient features of Opus and its implementation, emphasizing the issues related to the coordination of data-parallel HPF programs in a heterogeneous environment.

1 Introduction

Opus is a coordination language based upon HPF. Its central concept is the *shared abstraction (SDA)*, which generalizes Fortran 90/HPF modules using an object-oriented approach and imposing monitor semantics. The semantics of HPF data mapping directives and the ON clause are extended to allow the specification of resource allocation.

The primary motivation for the design of Opus came from the need to coordinate the execution of tasks in a multidisciplinary application (MDA) at a high level of abstraction. With the advent of teraflops supercomputers MDAs are increasingly used to deal with complex simulations, such as aircraft design optimization or climate modeling, where a number of interacting engineering disciplines must be coordinated. While each individual discipline is typically executed as a data-parallel SPMD module on a parallel supercomputer or a homogeneous cluster of workstations, the whole application runs in a heterogeneous environment. In particular, different modules may have different computational needs: while the code of one module may be specifically suitable for vector computers, other modules may need to exploit thread parallelism provided by SMPs. Thus, a language supporting such application not only needs to provide efficient coordination support, but also support for heterogeneous platforms. As a consequence, Opus is designed to support the

*The work described in this paper was partially supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund and by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19840, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

integration of data and task parallelism within a heterogeneous environment.

In this paper, we first outline the environment in which Opus programs are expected to run, focusing on the requirements posed by MDAs. This will be followed by an overview of a few other approaches that address this issue (Section 2). After a short description of the Opus language (Section 3), we will discuss the implementation of the language in Section 4, dealing also with the issue of interfacing with Java and communication libraries. The paper ends with concluding remarks in Section 5.

2 Requirements and Support for Multidisciplinary Applications

In this section we first shortly discuss the requirements posed by the salient properties of multidisciplinary applications and the computational environments in which they are executed. Based on these criteria we look at a number of languages and systems that can be used to address these issues, including Linda, Agora, CC++, Orca, Corba, Legion, and Globus.

2.1 Requirements

The target environment considered in this paper is a heterogeneous distributed network of computation nodes, each of which may be a single workstation, a cluster of workstations, or a parallel supercomputer. Multidisciplinary applications are typically developed by a team of programmers including experts from the individual disciplines. The discipline codes are largely autonomous modules, which are combined via well-defined interfaces. Once the interface has been specified, the individual modules can be developed independently; data accesses and method calls across interfaces must be tightly controlled to enforce privacy and security requirements.

MDAs exhibit at least two levels of parallelism. The discipline codes are typically computationally intensive data-parallel codes, executed on a homogeneous parallel architecture subject to the *Single-Program-Multiple-Data (SPMD)* paradigm. In contrast, the outer level of the application can be viewed as a set of coarse-grain tasks operating in a heterogeneous environment. In order to achieve the required performance, a language supporting such applications must allow the full exploitation of both levels of parallelism.

2.2 Related Approaches

We will first take a look at four languages that address the issues related to MDAs in different ways: Linda, Agora, CC++, and Orca. Subsequently, we will discuss three software architectures supporting heterogeneous environments. From a huge set of approaches we have chosen three representative projects: Corba, Legion, and Globus.

2.2.1 Linda

A Linda [1] execution operates in a global associatively addressed memory called the *tuple space* which can be accessed via atomic operations for adding, removing and reading of tuples. A special *eval* operation, when applied to a tuple of expressions, generates a set of parallel processes, one for each field of the tuple. The tuple space provides an uncoupled communication paradigm, covering communication in space as well as in time: a tuple remains in tuple space as long as it is not removed, independent of the lifetime of the process that created it.

Although these features provide some basic facilities relevant to MDAs, there is a lack of support for modular design, data parallelism, and standard data structures. Moreover, because of the fact that producer and consumer of a tuple are disconnected, the optimization of data transfers is not easily possible.

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

2.2.2 Agora

Agora [6], developed around 1986, allows the specification of *shared data types (SDTs)* in a Lisp-like syntax. The description of an SDT consists of the specification of its data structures, the associated set of methods, and an addressing mode. Agora provides a set of built-in methods (create, destroy, read, write, atomic execute), which can be combined into more complex user-defined methods. SDT objects can be addressed either by using indexing in a linear array scheme, or through the hashing of strings in a hash table.

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

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

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

2.2.3 CC++

Compositional C++ (CC++) [9] is a set of extensions for the C++ language. CC++ provides *processor objects*, which are instances of specially designated "global" classes that are associated with a unique address space and can be explicitly mapped to physical processors. One or more processes can execute on a processor object; they have access to all data in its address space.

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

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

CC++ allows multiple processes to execute in parallel inside a processor object. As a consequence, synchronization must be programmed explicitly where needed, using a single-assignment *sync* variable.

CC++ features such as processor objects, global pointers, and the synchronization facilities are highly useful for MDAs. However, the lack of asynchronous remote procedure calls and missing support for data parallelism and its integration with task parallelism reduce the applicability of the language for this class of applications. Recently, a consortium of researchers has proposed HPC++ as an attempt to overlay data parallelism and data distribution facilities on top of the explicit tasking facilities of CC++.

2.2.4 Orca

Orca [3], which was one of the precursors of Opus, provides an object model very similar to the model used by the Opus language described in the following section. However, Orca is a distributed programming language which – in contrast to Opus – has not been designed with the explicit objective of supporting the requirements of MDAs.

2.2.5 CORBA

One way of supporting portable and interoperable applications which has recently gained much attention is the *Common Object Request Broker Architecture (CORBA)* [25] as defined by the *Object Management Group (OMG)*. The central part of CORBA, the *Object Request Broker (ORB)* acts as a mediator between clients asking for a service provided by a server. All interaction is handled by the ORB; thus client and server are not directly connected and do not need to know details about their counterpart such as the physical location or the implementation language. Using a CORBA specific *Interface Definition Language (IDL)* a clean interface can be provided for all required services.

CORBA supports heterogeneity in both architecture and software. A CORBA service may be implemented in any language for which an IDL binding exists. Currently, languages for which IDL bindings are defined include C, C++, SmallTalk, COBOL, ADA, Java, but not Fortran.

Although CORBA provides a clean architecture for coupling different modules and for exploiting heterogeneous systems it has some weaknesses when taking the requirements of MDAs into account:

- CORBA does not provide explicit support for data parallelism
- Due to the generality of the ORB, interaction may be expensive
- CORBA does not provide adequate support for high level coordination of tasks

Recently, extensions to CORBA for high performance computing have been discussed in [17].

2.2.6 Legion

Legion [19] is a software architecture that supports a single, coherent virtual machine on top of the heterogeneous structure exhibited by the Internet. The key concept of Legion is a distributed object model. Objects can be distributed over the available computational resources in a user guided way; object interaction takes place via method invocation. Legion's vision is to use millions of hosts all over the world as one powerful supercomputer. Hence the design of Legion tackles a number of problems concerned with this vision such as fault tolerance, resource heterogeneity, security, language interoperability, scalability, and site autonomy.

Moreover, Legion also provides support for parallelism in that either parallel languages like MPL, BFS, or Java are supported directly, or that other parallel modules (like HPF or MPI codes) can be wrapped with an object and used within Legion. Similar to CORBA, Legion supports language interoperability by using IDLs.

Legion's object model is highly useful for MDAs, however it lacks efficient support for interaction among parallel objects. Due to the wrapping mechanism parallel objects appear as simple sequential ones to other Legion objects which prevents e.g. direct communication between all components of such objects. Moreover, Legion's support for parallel libraries like MPI may also cause performance problems: Legion requires such libraries to be emulated using the underlying Legion runtime library. Hence, vendor optimized implementations can not be exploited.

Although Legion has the notion of asynchronous remote procedure calls, such procedure calls can not be coordinated in a high level fashion.

2.2.7 Globus

The Globus [16] system provides a metacomputing toolkit which enables computing in distributed heterogeneous environments. Globus provides a set of services from which application developers can select to meet their needs. This set of services currently consists of 7 components:

- Resource management: resource allocation and process management.
- Communication: unicast and muticast communication services provided by Nexus.
- Information: distributed access to structure and state information.
- Security: authentication and related security services.
- Health and status: monitoring of health and status of system components.
- Remote data access: remote access to data via sequential and parallel interfaces.
- Executable management: Construction, caching, and location of executables.

While these services provide some useful functionality for the designer of an MDA, Globus does not offer the user a uniform programming and language model supporting the specific properties of such applications.

3 Overview of the Opus Language

Opus is an extension of High Performance Fortran (HPF). It provides a hybrid programming model for expressing multidisciplinary applications centered around the concept of the *Shared Abstraction (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.

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. Resource allocation in Opus is controlled by an "on-clause" which can be used to specify the machine and the number of processors on which the SDA should be created. Thus, the programmer can take advantage of the fact that some architectures are better than others at executing particular problems (e.g., vectorizable codes) and map SDAs accordingly. Moreover, data locality among SDAs can be exploited by specifying whether an SDA should use the same or a disjoint processor set as its creator. SDA variables are handles through which SDAs are accessed from within a program.

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

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

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

We use HPF syntax to support data parallelism within an SDA. Thus, each SDA type definition may contain an optional processors statement, as defined for HPF procedures [21]. This allows the internal data structures of the SDA to be distributed across the processors using the distribution and alignment directives of HPF. The dummy arguments of the SDA methods can also be distributed using the rules applicable to procedure arguments in HPF.

The execution of an Opus program can be thought of as a system of SDAs in which an SDA executes a method in response to a request from another SDA.

A more detailed discussion of the Opus language design as well as some application examples can be found in [11]. The full language specification is available in [10].

Sidebar: Practical Opus Programming

To illustrate the Opus programming paradigm as well as the efficiency of our prototype implementation we discuss a short example program that benefits from integrating task and data parallelism. Rather than discussing a full MDA, which goes beyond the scope of this paper, we present an Opus implementation of the Narrowband Tracking Radar (NTR) benchmark which was developed at MIT Lincoln Laboratories to measure the effectiveness of multicomputers in processing their sensor-based applications. This application, and much of its description, is extracted from the CMU Task Parallel Program Suite [14]. The Opus implementation was first discussed by Bal and Haines in [2] where they compare several languages that combine task and data parallelism, with respect to the NTR example.

The input for the NTR program is data from a sensor along $c = 4$ independent *channels*. For each channel, $r = 10$ vectors of $d = 512$ complex numbers are received which form together an $d \times r \times c$ matrix M. Such a matrix is received every 5 milliseconds. The NTR program performs three transformations on the input data: (1) r independent d-point FFTs are performed on all 4 channels; (2) for each of the channels, a sum is computed based on the scaled magnitudes of certain elements from all 10 vectors; and finally (3) each vector element is matched against a threshold, which is a function of the sums computed above and the element itself. Since the amount of data parallelism in this application is restricted by the parameters c, d, and r, which are given by nature and current sensor technology, task parallelism needs to be exploited when increasing the number of processors.

Figure 1 illustrates the NTR problem encoded in Opus. Note that due to space limitations only the overall structure of the program is shown and many details, especially the I/O, are considerably simplified. The computation tasks are encapsulated in the SDA TYPE `radar_type` which receives the complex matrix `M` constructed from the sensor data. The third dimension (the channels) of `M` is distributed using a `block` strategy. Thus, at most 4 processors can be efficiently used for the computation. Exploiting more processors by distributing also the 10 vectors does not lead to a significant performance gain, as can be seen in the performance measurements for an equivalent HPF version presented below. This is in large part due to the additional communication overhead introduced by the reduction operation for computing the sum of magnitudes in step (2). Data parallelism is exploited in the computation in that the `i`-loop over the channels is marked as `INDEPENDENT` and its iterations are distributed according to the distribution scheme of matrix `M` which is enforced by the `ON HOME` directive.

The main program first allocates a set of SDAs where the total number is dependent on the number of available processors. Each of these SDA objects is created on 4 processors of the Quadrics CS2 (specified in the `ON`-clause). The input data is then passed to the SDAs in a round-robin fashion by invoking the `compute` methods in an asynchronous way. Thus, all SDAs are allowed to work concurrently on disjoint data sets.

We implemented the NTR problem and compiled it using our prototype Opus compiler which transforms the Opus program into an HPF program according to our implementation design presented in Section 4. VFC [5] was used to compile the data parallel parts of the Opus code as well as a pure data parallel HPF version of the problem for comparison purposes. The HPF version makes use of a 2-dimensional processors array to which matrix `M` is distributed applying a `block` strategy for both the second (the vectors) and the third dimension (the channels):

```
!HPF$ PROCESSORS P(MAX(1,NUMBER_OF_PROCESSORS()/4),MIN(4,NUMBER_OF_PROCESSORS()))
!HPF$ DISTRIBUTE (*,BLOCK,BLOCK) ONTO P :: M
```

The runtime results (normalized to the sequential runtime) on the Quadrics CS2 are plotted in Figure 2. It can be seen that the data parallel implementation scales well up to 4 processors but does not improve if more than 8 processors are being used. Therefore, we designed our Opus implementation such that every SDA is created on 4 processors. The total number of SDAs equals the number of available processors divided by 4. By using Opus we can achieve a much better scaling behavior. In fact, the program scales almost perfectly up to 32 processors. As a consequence, the total runtime on more than 8 processors is substantially improved with respect to the data parallel one. For instance, the Opus program spends only about 25% of the data parallel runtime on 32 processors.

```
      SDA TYPE radar_type
!HPF$ PROCESSORS P(4)
      CONTAINS
        SUBROUTINE compute(M)
          COMPLEX M(512,10,4)         ! the matrix
          REAL magnitudes(512,10,4)   ! magnitude array
          REAL sum                    ! the sum of magnitudes
!HPF$     DISTRIBUTE(*,*,BLOCK) ONTO P :: M, magnitudes

!HPF$     INDEPENDENT, NEW(sum)
          DO i = 1,4
!HPF$       ON HOME(M(:,:,i)), RESIDENT(M(:,:,i))
            DO j = 1,10
              CALL FFT(M(:,j,i))                               ! step 1
              magnitudes(:,j,i) = COMPUTEMAGNITUDES(M(:,j,i))   ! step 2
            END DO
            CALL sum = COMPUTESUM(magnitudes(:,:,i))           ! step 2
            CALL MATCHTHRESHOLD(magnitudes(:,:,i),sum)         ! step 3
          END DO
        END SUBROUTINE compute
      END SDA TYPE

      PROGRAM radar
        SDA (radar_type), ALLOCATABLE :: radar_sets(:)
        INTEGER :: i,j,iter, n_sdas

        n_sdas = GET_AVAILABLE_PROCESSORS()/4
      ! create n_sdas independent sdas
        ALLOCATE(radar_sets(n_sdas))
        DO i = 1, n_sdas
          CALL radar_sets(i)%CREATE() ON (MACHINE='CS2',PROCESSORS=4)
        END DO

      ! spawn n_sdas independent compute tasks
        DO iter = 1, N_ITERS, n_sdas
          DO j = 1, n_sdas
            SPAWN radar_sets(j)%compute(INPUTSETS(iter+j-1))
          END DO
        END DO
      END PROGRAM radar
```

Figure 1: Opus code for the NTR problem

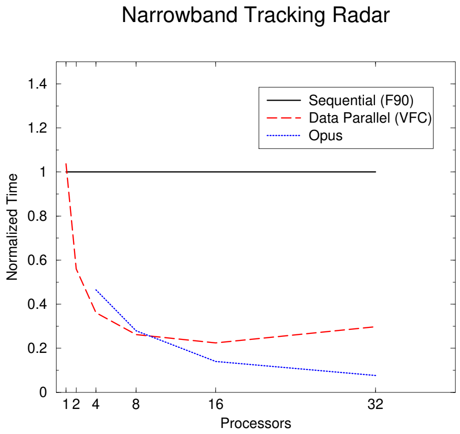

Figure 2: NTR Problem on the QWS CS2

However, the Opus implementation shows a significant overhead on 4 processors. This is in large part due to the additional data passing overhead. The Opus implementation needs to transfer a significant amount of data from the main program (where the data is produced) to the SDAs (where this data is processed). The data parallel implementation does not introduce such an overhead. Since we have only implemented a straightforward algorithm for doing the data passing among SDAs we expect the performance to improve when advanced redistribution libraries are integrated in our system.

4 Implementation Design

We recently initiated an effort for providing an efficient and portable Opus implementation. Our implementation design takes the exploitation of both homogeneous and heterogeneous environments into account. In particular, we are assuming a two-stage architecture model: an SDA is required to be created on a homogeneous platform (e.g. a DMMP or an SMP) which allows for an efficient exploitation of internal data parallelism. Different SDAs, however, may well reside on heterogeneous components of the network, thus enabling a mapping of specific computational tasks onto platforms which support them best. The Opus implementation takes care of providing the necessary means of interaction among such SDAs.

Another important design goal was the integration with an existing HPF compilation system (the *Vienna Fortran Compiler VFC* [5]) such that we do not have to re-implement any of the HPF compiler work and can focus on the coordination features of Opus. More specifically, we require the *Opus compiler* to transform the Opus source code in an equivalent HPF program, which is subsequently compiled by VFC.

In order to provide a flexible and portable implementation which also integrates well with the existing VFC runtime system, our design only requires a thin runtime layer built upon standardized packages for multithreading (e.g. pthreads) and communication (e.g. MPI). Making use of a multithreading model allows an efficient mapping of SDAs onto the physical resources as well as explicit scheduling, which is not possible if SDAs are mapped to processes instead of threads. Note however, that this requires all other components, in particular the MPI implementation, to be thread-safe.

Hereafter, we briefly present some of the key issues of our design for implementing SDAs within a homogeneous environment. A detailed discussion is out of the scope of this paper but can be found in [22]. Issues in heterogeneous systems are discussed afterwards.

4.1 Homogeneous Implementation Design

An Opus program is transformed by the Opus compiler into a set of subroutines which implement the SDA semantics. The structure of these subroutines is independent of the actual input program and is fixed in so-called *compilation templates.* These templates are filled in by the compiler using the specific application code.

An SDA is compiled to a separate thread enabling an efficient mapping of multiple SDAs to the same physical resources. However, within an SDA there is the need for multiple threads as well because we need a component that is able to receive method invocation requests (MIs) independently of ongoing method executions to avoid unnecessary blocks of the caller. Hence, one can think of a hierarchical thread structure where an SDA thread is subdivided into two independent threads communicating via a shared memory segment. As shown in Figure 3 the three components of an SDA object are:

- A shared memory area in which MI requests are stored in so-called *MI queues.* There are two such queues within every SDA, an *inqueue* for storing received requests and an *outqueue* for keeping track of outgoing requests. Recall, that methods may be guarded by a condition clause, thus the execution order of requests does not necessarily need to be the same as the receiving order. Hence, MI requests need to be stored together with all input data necessary for the method execution in form of *execution records.*
- The *Server Thread* which receives MI requests and places the execution records into the inqueue.
- The *Execution Thread* which retrieves records from the inqueue, evaluates the associated condition clause and executes the respective methods.

Both threads execute asynchronously in parallel; the only synchronization required is a mutually exclusive access to the shared queues.

SDAs communicate with one another via **Method Invocation** requests which are replaced by the compiler as follows:

A new execution record (the runtime representation of a method invocation request) is created holding pointers to all the arguments of the method. This record is inserted into the outqueue. Dependent on

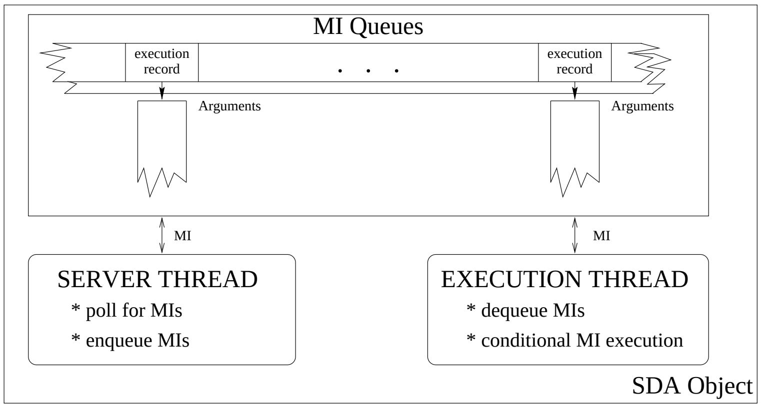

Figure 3: Structure of an SDA Object

the execution type (*synchronous* or *asynchronous*) a specific call to the runtime system is made asking for sending over the request. Subsequently, the needed input data is transferred to the callee and the caller resumes its execution immediately (in the case of an asynchronous call) or it will block until the MI has returned. However, it is guaranteed that the caller cannot resume its execution before all input data has been transferred. This is to ensure that data passed to a method cannot be in an inconsistent state.

After its creation, the **Server Thread** starts an infinite loop in which it is waiting for incoming MIs. If no messages are available, the underlying thread system will suspend the server thread. Having received an request it may either react directly (e.g. if another SDA inquires the state of particular method executions), or it marshals all arguments passed along with the request and puts the MI into the *inqueue* to be serviced by the execution thread.

Another task the server thread has to fulfill is the handling of *execution acknowledgments*: After having finished the execution of a method an SDA sends an execution acknowledgment along with possible results back to the caller. This acknowledgment is received by the server thread which checks if all the resulting data has been received and signals all SDAs waiting on the return of this request, allowing them to resume their execution.

The **Execution Thread** contains all of the method code, which is unaltered by the Opus compiler except for method calls, as well as the code required to evaluate the condition clauses. It also contains a generic "execution loop" that looks in the inqueue for work. The execution thread is implemented as a F90 subroutine whose body is the execution loop; the method procedures are compiled to internal subroutines as well as the condition clauses which are transformed to internal logical functions. Hence, all method-subroutines and condition clause-functions have access to the internal SDA data and the MI-queues via host association. The execution loop repeatedly checks the inqueue for enqueued execution records. If none are found, the execution thread is suspended. Otherwise, the procedure associated with the record is executed, given that the condition clause is satisfied; if it is not, another request stored in the inqueue may be served instead. The implementation of the execution thread guarantees that no two method procedures of an SDA can execute in parallel which is in conformance with the monitor-like semantics of SDAs.

Apart from SDA threads a special thread, the **Creation Thread**, is spawned at program start on every node in the Opus environment. This thread is in duty of *creating* new SDAs. When a new SDA needs to be *created*, the parent SDA sends a message to the *creation threads* of all the nodes specified in the on-clause. In response to this message the creation threads set up the new SDA object on these nodes. In particular, both the server and the execution thread are spawned and the shared memory area is allocated. In addition, a new communicator, containing all the nodes on which the SDA is created, is provided to the data-parallel VFC-runtime system which will use this communicator for its own communication needs.

Interaction between Distributed SDAs To allow for a data parallel execution, an SDA may be distributed across multiple processors. Thus, it consists of multiple threads. This fact gives raise to some non-trivial coordination and synchronization problems.

First of all we have to facilitate the exchange of possibly distributed data between SDAs which may well require data redistribution. Although we will use existing algorithms or libraries [26, 7, 12] for this task, we have to guarantee

1. that all nodes of both the caller and the callee agree in exchanging data, and
2. that the callee knows the actual layout of the input data in order to compute correct communication schedules.

The first problem is a synchronization problem among all the threads of an SDA. In order to properly synchronize them we need to combine them in a so-called SDA group (similar to the notion of ropes [20]) and introduce a master/worker relationship among them. The SDA master will notify its workers when data transfer is necessary such that all components of the SDA will work on the same problem.

The second problem can be solved by having the master of the caller SDA transfer the data layout descriptors of all procedure arguments to the master of the callee. The callee master notifies its workers in turn, broadcasting the layout descriptors. Note, that also descriptors for result arguments need to be transferred such that they are available for computing communication schedules when the results need to be sent back. There is no need for transferring layout descriptors from the callee to the caller since the caller has access to the internal representation of the callee which includes also layout descriptors for its procedures' arguments.

Having all nodes of the caller and callee agreed on performing the data transfer and having exchanged the necessary data layout descriptors, a redistribution library can be asked to compute communication schedules and to perform the actual data transfer. Note, that all the callee's communication takes place in the server threads; hence communication may well be overlapped by computation taking place in the execution thread.

Another synchronization problem is that all execution threads of an SDA need to work on the same method. Our MI request handling ensures the same ordering of execution records in all the inqueues of an SDA, thus, there is no need of extra synchronization between the master and its workers when selecting a record for execution. However, the evaluation of the condition clause may involve data replication if distributed data is needed for doing the evaluation. A possible alternative to having all execution threads of an SDA doing the evaluation would be to assign only the master with this task. In this case there is the need of synchronization between the master and its workers: the master has to signal which record should be executed.

4.2 Support for Heterogeneous Platforms

We have presented our implementation design assuming a homogeneous platform above. This design is able to make use of machine specific optimized components like thread and MPI libraries. Moreover, efficient data exchange among distributed SDAs is provided. However, as already mentioned, SDAs may have different computational needs and need thus to be executed on different components of a heterogeneous network. In the following we discuss some issues in exploiting heterogeneous platforms. Note, that with the term *heterogeneous platforms* we refer to networks where multiple (super)computers which do not support a common communication layer are interconnected. In particular, this view of heterogeneity does not cover clusters of heterogeneous workstations since e.g. the *mpich* or *LAM* implementation of MPI already provide a *parallel virtual machine* view of the cluster. Hence, heterogeneous clusters of workstations can already be efficiently exploited by our current Opus implementation. Of course, the programmer needs to be aware of possible slow connections between workstations when specifying the creation scheme of SDAs.

The main difference in SDA interaction on a heterogeneous network is that it is no longer possible to use sophisticated data exchange mechanism among distributed SDAs. Instead, the SDA master of the caller needs to collect all the data that is then sent to the SDA master of the callee which in turn distributes the data among its workers. This gather/scatter mechanism is needed because the nodes of different machines in the network are typically not directly connected but only one connection exists between the machines.

Apart from this change in the SDA interaction mechanism, additional runtime support for interconnecting machines efficiently is required. In the following we outline two promising approaches. Other possibilities like employing parts of the Globus toolkit (cf. Section 2.2.7) have not been considered yet but are within the focus of ongoing research.

4.2.1 MPI

The message passing standard MPI [23, 24] has been designed to operate correctly in heterogeneous environments. In particular, MPI requires every message to be explicitly typed, thus any needed data conversion can be applied. However, current MPI implementations for heterogeneous environments (e.g. *mpich* or *LAM*) consider only networks of heterogeneous workstations and do not tackle problems in combining different supercomputers. This restriction is largely due to the fact that many vendors provide proprietary MPI implementations optimized for their machines which do not match with each others. Recently, an industrial led effort was initiated which aims to provide an *Interoparable Message-Passing Interface (IMPI)* [13] enabling interaction among different MPI implementations. The main advantage of this approach is that every vendor may still implement its own MPI specifically tailored for his machine, while a standardized interface can be used to interconnect the different platforms.

Other efforts for providing MPI support for heterogeneous supercomputers include the PACX-MPI project [4] from the University of Stuttgart and the PVMPI project [15] from the University of Tennessee.

Using MPI as a base vehicle for implementing Opus on heterogeneous platforms is an appealing approach. Our design for homogeneous platforms assumes MPI as being the main communication library, thus we expect only minor changes in our runtime system when integrating e.g. IMPI. Moreover, the facilities of MPI2, especially the dynamic process creation support, are expected to be very useful in the context of dynamic resources management for Opus. However, it seems that current implementations are at a quite preliminary state which restricts their applicability. Nevertheless, we expect that IMPI (and MPI2) implementations will appear and mature in the near future; hence, we will keep our design open for their integration.

4.2.2 Java

Another possibility of interconnecting heterogeneous platforms is to exploit the networking facilities of the programming language Java [18]. Java provides a set of features which makes its use attractive in heterogeneous environments:

- *Platform independence:* once compiled, a Java program is expected to run on every machine for which a Java Virtual Machine (JVM) is available.
- *Common data representation:* in order to provide platform independence Java specifies a common data representation, thus there is no need for data conversion when transmitting data between heterogeneous platforms.
- *Communication features:* apart from low level socket communication Java also supports high level *Remote Method Invocation (RMI).* Moreover, data transfer is simplified since whole objects can be transmitted using *object serialization.*
- *Parallelism support:* the Java language has built in support for parallelism using a multithreading model.

An Opus implementation which uses the Java facilities for interconnecting heterogeneous platforms could especially exploit the object serialization features for transmitting SDA or method input data via sockets. We currently do not consider the usage of RMI techniques because they seem to be too high level for an Opus implementation. Moreover, the multithreading features of Java can be used to facilitate bi-directional connections among the machines. Using Java also enables the appealing possibility of providing predefined Opus services on the internet which can then be used within or similar to a bigger environment like e.g. NetSolve [8]. For such an approach issues in securing the access to SDAs need still to be clarified.

However, employing Java also has drawbacks, as compared to for instance using plain MPI (or IMPI):

- *Interfacing Java:* Interfacing Java from other languages is a difficult task. Currently, only an interface for C has been specified (the *Java Native Interface (JNI)*), therefore we need additional C wrapper functions for interfacing Opus with Java.

- *Efficiency:* Java programs tend to show a much lower performance as compared to equivalent C or even Fortran programs. This is in large part due to the additional layer of abstraction (the JVM).

Nevertheless, we believe that the deficiencies of Java as stated above do not prevent a prototype development: the efficiency problems of Java will not be prohibitive, since Java is only used as transportation vehicle and not for performing the computational task of an SDA. Moreover, just in time (JIT) compilers are expected to improve Java's performance significantly. The interfacing problem can also be solved by using C wrapper functions which can be generated by the Opus compiler.

More specifically, we intend to provide a layered Java program for managing inter-machine communication. On every machine which serves as a platform for an Opus application this Java program will be started when initiating the execution of the Opus program. In a multi-node machine the Java program will be started only on one node and all SDAs within the Opus program are aware of its location. The Java program consists of 3 layers: an interface to Opus (the *external server*), an interface to other machines (the *Java-Opus-Server*), and C wrappers (the *Opus-MI-Wrapper*) for interfacing F90 with Java. The Opus compiler generates all of these layers which have to fulfill the following tasks:

- The *Java-Opus-Server*, implemented in Java, is in duty of managing the low level communication between machines. It consists of two threads, one of these is managing incoming requests, the other sends outgoing requests to the callee. All communication is realized by using low level socket connection between a pair of Java-Opus-Servers.

- The *External Server* is the interface to the Opus program. It is implemented in F90 and runs as a separate thread within the Java program. Every SDA within an Opus program is able to communicate with the external server and all interaction between SDAs residing on different machines is managed by these servers. In fact, such SDAs do not interact directly with each other, but only with the external server.

- The *Opus-MI-Wrapper* is a set of C wrapper functions for SDA methods which are used to transform an Opus MI to Java and vice versa.

The Java-Opus-Server can be used to allow interaction between any Java application and SDAs. The Java application needs to make use of a specific (yet to be defined) *SDA class*, which is able to connect to an Java-Opus-Server and thus invoke methods in an SDA located on the same machine.

Note however, that this software architecture is only a preliminary design and we may well expect changes in the course of the ongoing implementation effort. Moreover, there is the need to carefully study the performance losses which may be involved by our architecture. However, we believe that the presented approach allows the rapid development of first prototypes which can be used to study the behavior of Opus applications on heterogeneous platforms in more detail.

5 Conclusion

Multidisciplinary applications typically exhibit multiple levels of parallelism and are often executed on a heterogeneous network of supercomputers. In this paper we presented the coordination language Opus which supports such applications. In particular, we focussed on an implementation design which allows an efficient exploitation of homogeneous components in the environment by using low level, optimized components. A more general approach needs to be applied for interconnecting heterogeneous components. Apart from presenting our implementation design we discussed a number of different approaches, both language and integrated tool support, which can be used for implementing MDAs.

A prototype implementation of Opus is being built and preliminary tests show promising performance. While the implementation design for homogeneous environments has already been specified, our design in the context of heterogeneous networks is still open, but we are considering the approaches discussed in this paper and are also studying alternatives, like employing parts of the Globus toolset.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *Computer*, 19(8):26–34, Aug. 1986.

[2] H.E. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3), 1998.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, Vol. 18(No. 3), March 1992.

[4] T. Beisel, E. Gabriel, and M. Resch. An Extension to MPI for Distributed Computing on MPPs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science. Springer, 1997.

[5] S. Benkner. VFC: The Vienna Fortran Compiler. *Journal of Scientific Programming*, 7(1):67–81, December 1998.

[6] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Trans. Computers*, 37(8):930–945, Aug. 1988.

[7] B. Carpenter, G. Zhang, and Y. Wen. NPAC PCRC Runtime Kernel (Adlib) Definition [Draft]. Technical report, Northeast Parallel Architectures Centre, Syracuse University, 1998.

[8] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3), Fall 1997.

[9] K.M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 218–313, Cambridge, Mass., 1993. MIT Press.

[10] B. Chapman, M. Haines, E. Laure, P. Mehrotra, J. Van Rosendale, and H. Zima. Opus 1.0 Reference Manual. Technical Report TR 97-13, Institute for Software Technology and Parallel Systems, University of Vienna, October 1997.

[11] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima. OPUS: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 6/9:345–362, Winter 1997.

[12] S. Chatterjee, J.R. Gilbert, F.J.E. Long, R. Schreiber, and S. Teng. Generating Local Addresses and Communication Sets for Data-Parallel Programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[13] IMPI Steering Committee. *IMPI - Interoperable Message-Passing Interface, DRAFT*. National Institute of Standards and Technology, August 1998.

[14] P. Dinda, T. Gross, et al. The CMU Task Parallel Program Suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March 1994.

[15] G.E. Fagg, J.J. Dongarra, and A. Geist. Heterogeneous MPI Application Interoperation and Process Management under PVMPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *LNCS*. Springer, 1997.

[16] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

[17] G. Fox and W. Furmanski. High Performance Commodity Computing. In I. Foster and C. Kesselman, editors, *The GRID. Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesely, 1996.

[19] A.S. Grimshaw, W.A. Wulf, et al. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), January 1997.

[20] M. Haines, P. Mehrotra, and D. Cronk. Ropes: Support for Collective Operations Among Distributed Threads. Technical Report ICASE-TR 95-36, ICASE, NASA Langley Research Center, Hampton, VA, 1995.

[21] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 2.0*, January 1997.

[22] E. Laure, M. Haines, P. Mehrotra, and H. Zima. On the Implementation of the Opus Coordination Language. Technical Report TR 99-03, Institute for Software Technology and Parallel Systems, May 1999.

[23] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 1.1*, June 1995.

[24] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[25] OMG. *CORBA/IIOP 2.2 Specification*, June 1998.

[26] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Proceedings of Frontiers '95*, pages 342–349, McLean, VA, February 1995.