On the Implementation of the Opus Coordination Language

Erwin Laure[1] **Matthew Haines**[2] **Piyush Mehrotra**[3] **Hans Zima**[1]

[1]Institute for Software Technology and Parallel Systems

University of Vienna

Liechtensteinstrasse 22, A-1090, Vienna, Austria

{erwin,zima}@par.univie.ac.at

[2]Department of Computer Science

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@icase.edu

Abstract

Opus is a new programming language designed to assist in coordinating the execution of multiple, independent program modules. With the help of Opus, coarse grained task parallelism between data parallel modules can be expressed in a clean and structured way. In this paper we address the problems of how to build a compilation and runtime support system that can efficiently implement the Opus con-

structs. Our design considers the often-conflicting goals of efficiency and modular construction through software re-use. In particular, we present the system requirements for an efficient Opus implementation, the Opus runtime system and describe how they work together to provide the underlying services that the Opus compiler needs for a broad class of machines.

1 Introduction

Data parallel languages, such as High Performance Fortran (HPF) [23], have been successfully applied to a wide range of numerical applications. For single programs, significant parallelism can be achieved using high-level constructs. However, many advanced scientific and engineering applications are multidisciplinary and heterogeneous in nature, requiring several independent programs working together to generate a solution. Such applications do not fit well into the data parallel paradigm, since there are no facilities for controlling the interaction between multiple, independent program units. The Opus coordination language [6, 7] was recently designed to fill this gap.

Opus defines language extensions to HPF that support the coordination of data parallel tasks. In doing so, Opus introduces coarse-grained *task parallelism* on top of an existing data parallel model, where each *task* is represented by an independent program module. Opus also introduces a mechanism to control the coupling of multiple programs into complex multidisciplinary codes. To achieve these goals Opus provides the following set of features:

- encapsulation of separate programs into modules in a way that respects their separate name spaces;
- coupling between modules at a high language level, as opposed to using explicit message-passing constructs throughout the code;
- task-level parallelism between modules and data parallelism within each module; and
- flexible and general synchronization mechanisms that allow the programmer maximal freedom in exploiting task-level parallelism.

In a previous paper [7], we presented the syntax and semantics of the Opus language. In this paper, we describe the design of a system that efficiently implements the Opus constructs. We provide an overview of the whole system, ranging from the low level virtual machine up to the high level compiler component. In designing our system, we have tried to satisfy several requirements, including efficient support for data parallelism, an efficient implementation of the coordination mechanisms of the language, the possibility to extend and modify the language, and portability to a broad class of machines.

Apart from describing our present system design, we discuss design alternatives, together with their advantages and disadvantages, and the motivation for our choices. Our approach deals with the often-conflicting goals of efficiency and modular construction through software re-use. Software re-use is of invaluable importance when designing a highly complex system. For example, we utilize an existing HPF compilation system (the *Vienna Fortran Compiler VFC* [3]) for the data parallel portion of our work. We also utilize existing libraries that provide useful communication and threading abstractions, but avoid larger runtime packages such as Nexus [13] and Chant [19] for reasons that we will explain later in more detail.

Finally, although our system has been specifically designed to meet the requirements of the Opus language, we believe that the design decisions described in this paper can be applied in a more general context. In particular, we show that the usage of a multithreaded virtual machine is advantageous for modeling complex asynchronously interacting tasks and we analyze the relationship between compiler and runtime system functionality.

The remainder of this paper is organized as follows: After a short introduction to the Opus language in Section 2 and the Opus execution model in Section 3, we present our detailed system design in Section 4. Possible design alternatives and related work are discussed in Section 5. In Section 6 we validate the effectiveness of our approach with some preliminary results. We end in Section 7 with concluding remarks and a discussion of future work.

2 The Opus Language

Opus introduces a small set of extensions to HPF [23] designed to coordinate an efficient parallel execution of multiple, independent data-parallel modules. At the heart of these extensions is an abstract data type called a ShareD Abstraction, or *SDA*, whose purpose is to provide a means for encapsulation of data and methods (procedures) which act on this data. Syntactically, Opus borrows heavily from Fortran 90 [26] in the definition and use of SDAs. SDAs may exploit data parallelism in that the internal data of SDAs as well as the data of SDA methods may be distributed using HPF data mapping features.

Viewed as an abstract data type, an *SDA type* specifies an object structure, containing *internal data* and the *methods* (procedures) which manipulate this data. By *creating* an instance of an SDA type an *SDA object* is generated. In the following, when it is clear from the context, we will simply refer to *SDAs* instead of *SDA objects*. At creation of an SDA, the resources on which it will execute are allocated, and all the internal data of the SDA is also allocated and initialized in order 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. During its *lifetime*, which is the time between its creation and its *termination*, an SDA can be accessed (a method of an SDA can be invoked) from within a program via *SDA variables*.

An SDA method can be invoked *synchronously*, where the caller is blocked until control returns, or *asynchronously*, where the caller does not have to wait for the completion of the method. Explicit synchronization is possible via *event* variables that can be associated with asynchronous method invocations. More specifically, an SDA may *test* (or poll) an event in a non-blocking fashion, thus getting information about a method's status without having to wait for its completion, or it may *wait* until a method has finished its execution. No two methods of an SDA are allowed to execute in parallel; thus each method has *exclusive access* to the data of the SDA. A method may have an associated *condition clause*, specifying a logical expression, which guards the method's activation.

In Figure 1 we illustrate the usage of Opus with the standard producer/consumer problem. In this example a set of producers and consumers operate independently, communicating and synchronizing each other via a bounded FIFO buffer. Note, that this example exploits only the task parallel features of Opus and does not make use of data parallelism.

*** Figure 1: Producer/Consumer Problem with Opus ***

3 The Opus Execution Model

In order to provide a better understanding of the implementation issues discussed in the following section we sketch the Opus execution model below, taking the above producer/consumer problem as a practical example.

During its execution an Opus program consists of a *main procedure* and a set of *SDA objects* which are created and destroyed dynamically during the lifetime of the main procedure. Every SDA (and the main procedure as well) executes asynchronously in a unique address space called *SDA Address SPace (SASP)*, containing the data and the executable code of an SDA. Each SASP is mapped to a set of physical computational units called *nodes*, where a node can, for example, be a workstation or a processor in a multi-processor machine. The distribution of a SASP across more than one node, for example the processors of a distributed-memory machine or a network of workstations, allows the exploitation of data parallelism. Distributed SASPs give rise to some difficult issues such as proper synchronization of method executions and data exchange as will be discussed in Section 4.4. A good overview of issues relating to the combination of task and data parallelism can be found in [1, 28]. In order to exploit the available resources efficiently, multiple SASPs may reside on a node.

The execution of an Opus program starts by executing the main procedure, so that we begin with a single (possibly distributed) SASP. When new SDAs are created, we allocate new SASPs on specific sets of nodes. After an SDA has been created, it runs asynchronously and in parallel with all other SDAs in the program. There is a parent/child relationship between SDAs; a parent SDA can only terminate if all of its children have finished their execution. SDAs communicate with each other via *method invocation (MI)*.

During its lifetime an SDA receives method invocation requests from other SDAs which are buffered and, depending on the associated "when-conditions", one of these buffered request is chosen to be fulfilled, that is, the associated method is executed. During the execution of a method an SDA may communicate with other SDAs via MIs, it may create additional SDAs, or it may wait for the return of issued MIs.

In our producer/consumer example the *main procedure* creates np+nc+1 SDAs (1 *buffer*, *np producers*, and *nc consumers*). The *main procedure* issues an asynchronous MI to every *producer* (*consumer*), causing the execution of the *produce* (*consume*) method. During the execution of these methods synchronous MIs to the *buffer* are issued, resulting in the execution of the *put* or *get* method, respectively. The *buffer* executes these methods conditionally, thus ensuring correct program behavior.

An Opus implementation has to provide an efficient mapping of SASPs to low level services (like threads and processes) available on nodes as well as efficient means of interaction between SDAs. Moreover, it has to ensure that an SDA is capable of receiving and buffering new MIs independently of ongoing method executions. Buffering MIs should be accomplished without unnecessary copying of MI arguments, which can include large segments of data. We provide a detailed discussion of how an implementation of Opus can fulfill these requirements in the following section.

4 An Implementation of Opus

In this section we discuss in detail how an Opus program is compiled and describe the components of the Opus system. Apart from having an efficient and portable implementation one of the main design goals was the integration with an existing HPF compilation system (the *Vienna Fortran Compiler VFC* [3]), so that we do not have to re-implement any of the HPF compiler work.

4.1 Introduction

The transformation of an Opus source code into an executable program is a 3-stage process as can be seen in Figure 2:

- The Opus Compiler transforms the code into blocks of HPF code with calls to the Opus Runtime System (*ORS*). This transformation process uses *compilation templates* which specify code structures common to every Opus application (cf. Section 4.3).
- This code is further processed by VFC which produces Fortran 90 code with calls to the VFC runtime system.
- The resulting Fortran 90 code is eventually compiled with a Fortran 90 compiler and linked with both the VFC runtime system and the ORS.

*** Figure 2: The Opus Compilation Process ***

While in the initial implementation of Opus [20] a fairly sophisticated runtime system was used, in our present system design the role of the runtime system is reduced to basic communication and SDA management tasks (cf. Section 4.2), while the Opus compiler implements most of the SDA semantics such as scheduling of MIs and efficient argument handling. In particular, the ORS integrates and abstracts lower level system components for both, SASP mapping and interaction (cf. Figure 3). We require communication mechanisms that do not assume a shared memory model for realizing SDA interaction. For an efficient mapping of SASPs we need to facilitate mappings of multiple SASPs to the same node, in order to exploit locality between SDAs. Such a mapping could be realized on Unix-based systems by mapping SASPs to processes, but this approach suffers from two major drawbacks: the inability to control scheduling and the costly context switching overheads (cf. [20]). In light of the problems with using a process-based model, we have elected to use a *multithreading* model. However, some support for a process-based model is necessary for platforms which do not allow multithreading as will be discussed in Section 4.5.

In our current design we make use of low-level packages for both communication and multithreading. The actual modules used vary from system to system, but include systems like MPI [29], TCP/IP [24], and POSIX threads [25]. These low level components can be seen as part of the virtual machine on top of which an Opus application will be executed.

We have taken a simplified approach to resource management: we presume that all the required resources are statically allocated. The extension of our design to support dynamic acquisition of new resources is subject to ongoing work.

We now provide a detailed discussion of the Opus system design, in particular of the Opus Runtime System (*ORS*) and the Opus Compiler.

4.2 The Opus Runtime System

The Opus Runtime System (ORS) implements the low-level runtime support that is utilized by every Opus application. Specifically, the ORS provides runtime routines for SDA management, including initialization, finalization, and MI handling, which are exploiting the functionalities provided by the underlying systems components (cf. Figure 3).

*** Figure 3: The Opus Runtime System and its Supporting System Components ***

Due to the conditional execution of methods it is necessary to buffer MIs on the callee side until the method is eventually ready for execution. This buffering is a critical task since MIs may be accompanied by large amounts of data being passed as input arguments to the method. MI-buffers are realized using two FIFO queues called *MI-queues*: the *inqueue* buffers incoming MIs while the *outqueue* keeps track of issued MIs. The inqueue management ensures some fairness in processing MIs by choosing the next earliest MI in the list after each unsuccessful test of a "when-condition".

MIs are represented in the form of *execution records* which contain information about the caller and the callee, the method that is to be executed, and a set of marshaled arguments. Figure 4 shows the type declaration (using Fortran 90 syntax) of an execution record, where the derived type `ors_sda_handle` is the runtime representation of SDAs and the type `ors_generic_type` is a compiler generated abstraction of the method arguments; apart from these pointers the execution record holds a unique `id`, the `id` of the method to be executed, the type of the execution (synchronous or asynchronous) and a logical flag which provides information about the execution progress. The `action` component allows the use of execution records not only for MI requests, but also for inquiries about the state of an MI and execution acknowledgments (cf. Section 4.3). The ORS implementation takes care that the management of input data does not involve any data copying that is not caused by the underlying message passing system, hence an execution record has only a pointer to the method arguments.

*** Figure 4: The Execution Record ***

The ORS also supports argument marshaling that is necessary for proper MI execution. The complex nature of SDA methods and the fact that the internal data structures can be distributed over multiple nodes makes argument marshaling a non-trivial task that may involve data redistribution. We make use of existing redistribution algorithms or libraries [5, 8, 30] for this purpose. In implementing the ORS, we were careful to avoid the underlying message passing system when communicating between two SASPs co-located on the same node, and used direct memory copies in these situations.

Apart from basic MI support, the ORS also facilitates SDA synchronization like the event mechanism described in Section 2. It also provides a higher-level interface to the underlying system components (e.g., creating a new thread or synchronizing threads via mutexes and condition variables), so that these packages can be easily exchanged without affecting the ORS or the compiler.

In the following we provide a more detailed discussion of the main ORS routines whose interface is given in Figure 5. The use of these routines can be seen in the following section where we discuss the Opus Compiler and its compilation templates. We omit the presentation of threading routines since those are simple wrappers around the low level routines. It is worth noting that our actual implementation of the ORS is done in Fortran 90 which facilitates the integration with existing HPF-runtime systems as will be discussed in more detail in Section 5.

*** Figure 5: Main ORS Routines ***

- `ors_initialize_sda(iq,oq)`: The initialization routine of an SDA. It allocates the *inqueue* `iq` and *outqueue* `oq`.
- `ors_terminate_sda(oq)`: This routine is invoked on the termination of an SDA and ensures correct behavior on termination. It is necessary to wait on the return of all outstanding MIs (which can be accessed via the outqueue `oq`), and a termination signal is sent to all children of the SDA. The routine has to wait until all of the SDA children have terminated in turn. After that it is safe to delete the SASP and to return.
- `ors_get_rec(r, queue) result(return_rec)`: Depending on whether the optional argument `r` is available, a pointer to the earliest execution record ready for processing or a pointer to the record matching `r` is returned. The latter case arises when the execution progress of an MI is queried by the intrinsic "wait" or "test" procedure. If `queue` is empty, the call will block until a new record has been inserted.
- `ors_insert_rec(r, queue)`: A new record is inserted into the queue.
- `ors_delete_rec(r, queue)`: A record is deleted from the queue.
- `ors_new_event(e,r)`: This routine binds an event variable to an execution record.

- `ors_test(e,s) result(finished)`: This routine facilitates a *test* inquiry, where `s` is the SDA-handle of the caller. A blocking MI is sent to the owner of event e (which is the SDA that has issued the MI to which the event is bound). The result of this MI is a logical flag indicating the execution progress of the MI bound to the queried event.

- `ors_wait(e,s)`: This routine is similar to `ors_test` except that it facilitates a *wait* inquiry. Thus, it will only return if the MI bound to the queried event has finished its execution. The counterpart of both routines, `ors_test` and `ors_wait`, on the callee side is the routine `ors_handle_inquiry`.

- `ors_handle_inquiry(r,c,a)`: This routines handles an inquiry `a` (which may be a *test* or a *wait*) on a specific asynchronous MI referred to by `r`. `c` is the execution record of the inquiry. In case of a *test* the *finished* component of `r` is immediately returned to the caller. In case of a *wait*, the caller is blocked until the the queried MI returns.

- `ors_handle_return(r)`: This routine is invoked when an SDA receives an execution acknowledgment for an issued MI. It checks whether all the results have been received, cleans up the outqueue (i.e., the execution record is deleted) and informs all SDAs waiting on the return of the MI (i.e., all SDAs which have issued a *wait* on the event bound to the MI) that it is finished. Note, that records of asynchronous MIs cannot be deleted completely; some minimal information (including the *finished* flag and the *ID*) needs to be maintained because wait- or test-inquiries may still occur.

- `ors_receive_mi(r)`: This is a wrapper for the message passing calls required for receiving MIs. Note that these message passing calls should be blocking, such that a thread which is waiting for an MI is suspended giving other threads the possibility to perform useful work. The routine allocates a new execution record and receives all the record data from the caller, except the input arguments. The calls for receiving input arguments are generated by the compiler as will be discussed in the following subsections.

- `ors_send_synchronous_mi(r)`, `ors_send_asynchronous_mi(r)`: These two routines are the counterparts of `ors_receive_mi(r)` on the caller side. An execution record is sent over to the callee; again, no input arguments are communicated. See Section 4.4 for a detailed discussion of input argument transfers.

4.3 The Opus Compiler

The last component of the Opus system is the Opus compiler. The compiler restructures the Opus code and produces blocks of code that utilize the functionalities provided by the ORS. More specifically, the compiler translates the code of an SDA into a set of subroutines which implement the SDA semantics. The structure of these subroutines is independent of the actual input program and is embodied in what are called *compilation templates*. These templates are filled in by the compiler using the specific application code.

The Opus compiler has been designed so as to work in conjunction with an HPF compiler such as the *Vienna Fortran Compiler (VFC)* [3]. As already shown in Figure 2, the input program is transformed into an HPF-conforming program with calls to the ORS. This HPF program is subsequently compiled by VFC, producing a Fortran 90 program with calls to both the ORS and the VFC runtime systems. By making use of both, the Opus compiler and VFC, we can specifically focus on the tasking issues of Opus and allow VFC to handle the data parallel portions. Therefore, the compiler needs only to restructure SDA code, and leaves the remainder of the code as well as all the internal code of SDA methods which is not concerned with MIs to be handled by VFC.

As already discussed in Section 4.1, the mapping of SASPs onto nodes can be *process based* or *thread based.* In the following we assume a *thread based* mapping and give some remarks on a process based implementation in Section 4.5.

By compiling SDAs into separate threads multiple SASPs can be efficiently mapped to the same physical resources. However, there is the need for multiple threads within an SDA as well because we need a component that is able to receive MIs independently of ongoing method executions. 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 6 the three components of an SDA are:

- A shared memory area, in which the *MI-queues* as discussed in Section 4.2 are allocated.

- 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 "when-conditions" and executes the respective methods.

*** Figure 6: Structure of an SDA Object ***

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

Apart from the internal threads of each SDA, a special thread called the *creation thread*, is spawned on every node at the start of the program. When a new SDA needs to be *created*, the parent SDA sends a message to the *creation threads* of all the nodes on which the SDA is to be established. In response to this message, the creation thread on each node sets up a new SDA object on its node. 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 has been created is provided to the data-parallel VFC-runtime system which will use this communicator for its own communication needs. In the following, we describe the compilation of method invocations and introduce the compilation templates for both the server and execution thread.

Method Invocation SDAs communicate with each other via method invocation requests and synchronize by *waiting* or *testing* an event associated with the invocation. Such SDA interactions are implemented by replacing the respective Opus calls with calls to the ORS according to the compilation template shown in Figure 7.

The first step is the creation of a new execution record. Note, that the argument component of the execution record also holds pointers for results and arguments with intent OUT or INOUT, thus the routines for receiving the results can make use of these addresses and do not need to make additional copies. This record is then inserted into the outqueue[1] and, if an event was specified, the event variable is bound to this record. The actual invocation is initiated by making a call to the ORS and the needed input data is transferred to the callee. In the case of an asynchronous call, the caller resumes its execution immediately

[1] The outqueue is accessible to a method via *host association* as will be explained when discussing the execution thread.

while in the case of an synchronous call it will block until the MI has returned. However, in either case, the caller does not resume its execution before all input data has been transferred in order to ensure that the input data to the invocation is not modified during the process of setting up the MI. A call to the intrinsic routines *wait* and *test* are simply compiled to a respective ORS calls.

*** Figure 7:Compilation Template for Method Invocations ***

The Server Thread Although the primary task of the server thread is to receive and enqueue MIs while the method executions are carried out by the execution thread, it also directly responds to inquiries about the state of an MI. This allows such queries to be "short-cut" and answered immediately, rather than being enqueued for service by the execution thread. Such queries occur within the context of event management (*wait* and *test*).

Figure 8 shows the compilation template for a server thread. Once the server thread is created it starts an infinite loop in which it is waiting for incoming MIs. This call is blocking and the underlying thread system ensures that the server thread is suspended if no messages are available. After having received an MI, the server thread takes the appropriate steps dependent on the *action* that has to be performed in response to the MI. The server thread is able to distinguish between four different actions:

- *Inquiries:* A *wait* or *test* request is received which relates to an event that is bound to an MI located in the outqueue. The server thread can serve this request immediately without involving the execution thread by looking up the outqueue for the matching execution record and subsequently calling the ORS to handle the request.
- *Execution Acknowledgment:* An acknowledgment from another SDA is received which indicates that an MI has finished its execution. Similar to inquiries, first the matching execution record has to be retrieved from the outqueue and then the ORS is invoked in order to handle the return.
- *Method Call:* A request for executing an MI is received. Depending on which method should be executed the server thread receives all the input arguments and adds them to the execution record which in turn is inserted into the inqueue. It is guaranteed that all input arguments have been received before the execution record is inserted into the inqueue and thus becomes available to the execution

thread. Note, that all communication carried out by the server thread may well be overlapped with useful computation in the execution thread.

- *Termination:* This flag indicates that the server thread should terminate.

The compiler has to fill in the appropriate calls to the ORS for receiving the arguments of MIs. By compiling the exchange of actual arguments we can avoid additional copying that would occur if the ORS has to receive the arguments without knowing the actual procedure interfaces.

*** Figure 8: Compilation Template for the Server Thread ***

The Execution Thread The execution thread (cf. Figure 9) contains all of the method code, which is unaltered by the Opus compiler except for the method calls, as well as the code required to evaluate the "when-conditions" for conditional SDA method invocation. The execution thread also contains a generic "execution loop" that looks in the inqueue for work. The execution thread is implemented as a Fortran 90 subroutine whose body is the execution loop. The method procedures are compiled to internal subroutines and the "when-conditions" are transformed to internal logical functions. Hence, all method-subroutines and "when-condition"-functions have access to the internal SDA data and the MI-queues via host association. The execution loop repeatedly checks the inqueue for MIs. If none are found, the execution thread is suspended. Otherwise, the procedure associated with the MI is executed, given that the associated "when-condition" function evaluates to true. 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.

In particular, the execution thread performs the following actions:

- After a call to the ORS in order to initialize the SDA (in particular the outqueue and inqueue), the execution thread enters the *execution loop.*

- In the execution loop the inqueue is checked for MIs. If there is an execution record available, the *when* function matching the indicated method (identified by the `method_id` component of the execution-record), is evaluated. If the "when-condition" is satisfied, i.e., the function returns true, the MI is processed as follows:

- The associated method is executed.
- The ORS is asked to send back an execution acknowledgment and the results.
- Finally, the record is deleted from the inqueue.

*** Figure 9: Compilation Template for the Execution Thread ***

4.4 Interaction between Distributed SDAs

To allow for data parallel execution, an SDA may be distributed across multiple nodes. Thus, depending on the underlying system, an SDA consists of multiple threads or processes. This fact gives rise 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. As already noted, existing algorithms or libraries [5, 8, 30] will be employed for this task. However, in order to make use of these libraries, we have to ensure that:

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

The first issue above is a problem of synchronizing all the threads/processes of an SDA. In order to do this properly, we need to combine them in a so-called SDA group (in a multithreaded context this is similar to the notion of ropes [21]) 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 transfer.

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, in turn, notifies its workers, broadcasting the layout descriptors. Note, that the descriptors for the result arguments also 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 also includes the 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. Figure 10 illustrates the data exchange process. Note, that all the communication takes place in the server threads; hence communication may well be overlapped by computation taking place in the execution thread.

Another synchronization issue is that all execution threads of an SDA need to work on the same MI. The MI handling described above 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 "when-conditions" 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 then the need of synchronization between the master and its workers: the master has to signal which record should be executed.

*** Figure 10: Illustration of the MI Process for Distributed SDAs ***

4.5 Practical Remarks on the Implementation

We have presented a detailed design description of the Opus system above. However, the actual implementation needs to deviate from this "ideal" design to some extent because of interoperability problems between some of the system components as we discuss below.

Thread Compliant Message Passing Systems Our design, as described above, assumes a multi-threaded system. This requires the underlying message passing system to be thread compliant as specified in the MPI-2 standard [29]. However, current MPI implementations, such as *mpich* [16], are not thread compliant which forces some changes to our implementation.

In particular, we have to ensure that at any time there is only one MPI-call active on a node. We can accomplish this by assigning all the MPI calls to the *server thread* and having only a single server thread per node to serve all the SDAs on the node instead of having a separate server thread for each SDA. This shared server thread then can also carry out the responsibilities of the creation thread. One of the issues that arises is that the server thread cannot block on a call to the `ors_receive_mi` routine, since it is servicing calls for multiple SDAs. Hence, the server thread has to actively poll for messages which may cause a significant overhead.

Process Based Implementation If the underlying system does not allow a mapping of SDAs to threads, a process based mapping has to be utilized. Hence, an SDA cannot be internally multithreaded as described in our design. In particular, this means that there is only one thread of control which has to fulfill the tasks of both the server and the execution threads. This can be accomplished by merging the codes of the server and the execution thread. More specifically, the compilation template of the execution thread is used as basis for the new implementation and the code of the server thread template is included into the execution loop. This ensures that new MIs can be received if no method is being executed. The implementation of the routine `ors_receive_mi` has to be changed such that it is not blocking anymore but instead actively polls for MIs.

Another problem within the context of process based implementations is that the underlying message passing system must be able to support dynamic process management. Unfortunately, this is, for example, not possible with the current implementation of *mpich* [16]. An implementation based upon LAM-MPI [27] could be considered instead.

Heterogeneous Environments Even though our current implementation executes Opus programs in a homogeneous environment, there is nothing inherently in the design which prohibits execution in a heterogeneous environment. If the underlying message passing system supports heterogeneous networks, an Opus program will run without any modifications in the environment. We are currently investigating different possibilities for facilitating executions of Opus programs on heterogeneous platforms by using communication systems others than MPI in such environments such as the Nexus [12] library or the Java [15] networking facilities.

5 Rationale and Related Work

In this section, we highlight the alternative design possibilities for the different components of our system and provide reasons for our specific design choices. We also contrast our system design with other systems being developed for integrating task and data parallelism.

5.1 System Support

In Section 4.1, we identified the functionalities that the underlying support system has to provide for the ORS to work properly. The key requirements are *multithreading* and *communication*. Thus, we could have employed one of the packages that combine these features, such as Chant [19], Nexus [13], or Panda [4]. The advantage of using such packages would be that they provide a high level of abstraction for both multithreading and communication, often using an interface similar to the well known *pthreads* interface. With help of these packages, we can ignore the low-level details encountered when integrating multithreading and communication, such as direct communication between threads or proper thread synchronization. Moreover, most of these packages also provide support for a remote service request abstraction.

However, for an Opus implementation, as we have described in this paper, we felt that the disadvantages of using a high-level runtime system like Chant or Nexus outweighed the advantages of the features they offer. There are several reasons for this. First, many of the features provided by these packages, such as remote service requests, do not directly implement the proper semantics of SDAs, such as conditional execution of methods and distributed method arguments. Expressing the SDA semantics with these high level constructs often adds unnecessary overhead which can be avoided by making direct use of lower level systems. Second, because we have chosen to compile the SDA specifications as much as possible, the requirement for sophisticated runtime support is greatly reduced. Instead, only a small set of lightweight interfaces for threads and communication is desired. Third, there is a serious software engineering problem with integrating a large runtime package with an existing HPF runtime system such as VFC. This is due in large part to the inability of most runtime system packages to change their behavior in accordance with the needs of different users [18]. Finally, if all one needs is basic threads and communication, then using these higher-level runtime systems often just adds unwanted overhead to the performance of the operations. For example, although it is possible to access MPI calls from within Nexus [11], doing so is much slower than accessing the machine-specific MPI implementation directly.

In light of these disadvantages for using high-level runtime support systems, we decided to use low-level communication (e.g., MPI, TCP/IP) and multithreading (e.g., pthreads) packages directly as basis for our ORS implementation. By using these standardized packages we can easily integrate the ORS with the VFC runtime system, thus enabling us to port to another platform that supports VFC with little effort.

5.2 The Opus Runtime System and the Opus Compiler

The main component of the Opus system is the Opus Compiler which targets the Opus Runtime System. The compiler and the ORS are tightly interrelated hence we combine their discussions in here.

One of the major design decisions was where to put the borderline between the compiler and the runtime system. In previous work on the implementation of Opus [20], we assumed that the runtime system would provide a generic interface and manage most of the coordination between SDAs. The task of the compiler was only to insert calls to the runtime system where appropriate. This approach, which is commonly used to provide an initial implementation of a new language, assumed a fairly sophisticated runtime system based upon Chant [19]. However, this approach had some major drawbacks. In order to provide a generic interface to the compiler, the runtime system was supposed to be implemented in C/C++, since Fortran 90 does not support any abstraction for generic data types. This meant that all the data needed for method invocations had to be converted from Fortran 90 to C/C++, stored and managed there, and subsequently be converted back to Fortran 90. This conversion, however, is a non-trivial task since there are major differences in the storing schemes and argument handling between C/C++ and Fortran 90. Moreover, storage schemes and argument handling are not standardized in Fortran 90, thus, any attempt to interface Fortran 90 with C/C++ would be compiler dependent and not portable anymore. Another problem with that approach would be generating and using C-wrappers for all the HPF routines, resulting in additional calling overhead.

The same problems arise when using other high level coordination libraries like KeLP [9] which is also a C++ library that needs additional conversion of the HPF data and HPF mapping descriptors into its own convention.

Again, for the approaches discussed above interoperability problems with the VFC runtime system would occur, as described in the previous subsection.

The HPF binding of MPI (HPF/MPI [14]) could probably more easily be integrated with VFC. However, it has some problems in providing the flexibility that Opus needs since it is targeted more towards programmers rather than compilers.

Our decision to let the compiler do most of the work and using only a small runtime system written in Fortran 90 was mainly driven by the disadvantages in using languages other than Fortran 90 as described above. Although the task of the compiler is limited at the moment, in fact, it only fills in compilation templates, making extensive use of the compiler may have further benefits in the future. By using a compiler we can apply optimizations to avoid expensive runtime checks which cannot be skipped by a generalized runtime system. In addition, the compiler could recognize and exploit further potential parallelism like intra-SDA parallelism which is a major topic for future research.

Related coordination approaches that make use of a compiler are, for example, Fx [17], Fortran-M [10], or Orca [2]. Fx does not need any runtime support since it has a quite simple tasking model similar to the tasking facilities in HPF-2 (in fact, HPF-2 took over many ideas introduced in Fx). Fortran-M is based upon Nexus which fits perfectly into the Fortran-M model. However, Fortran-M is a purely task parallel language thus the problems of Nexus in a data parallel context do not arise. Similarly, Orca was initially designed as a task parallel language based upon Panda. Subsequently, Orca was extended to support data parallelism [22] using concepts similar to those in Opus. However, in contrast to Opus, Orca is not an extension of an existing data parallel language but a new language where both data and task parallel features have been designed in a way that they integrate well. Hence, Panda was designed to support both task and data parallelism whereas the ORS has to be integrated with an existing data parallel runtime system.

6 Implementation Status and Preliminary Results

We have implemented a prototype of the ORS on the MEIKO CS2 as well as on a cluster of Solaris Workstations. The implementation was mainly done in Fortran 90. The ORS is based on *mpich* [16] (on the Meiko) and on TCP/IP [24] (on Workstations), respectively, as well as on the native Solaris *pthreads* implementation. This prototype is currently restricted to the task parallel features of Opus.

Although the implementation of the compiler is not yet completely finished, we have hand-compiled a simple version of the producer/consumer problem (cf. Figure 1) to prove the effectiveness of our approach. In this simple example, we simulated one producer and one consumer acting on a bounded buffer. The preliminary runtime results of this hand-compiled program can be found in Figure 11(a) for the Meiko and in Figure 11(b) for the Workstation-Cluster.

*** Figure 11 (a) and (b): Preliminary Results for the Consumer Producer Problem ***

From the results for the Meiko, it can be seen that our implementation introduces only a small overhead to the sequential version using one processor. We can achieve almost perfect speedup onto two processors, whereas using more processors does not lead to any additional performance gain. This is due to the internal parallelism of the problem which is restricted because we simulate only one producer and one consumer.

The situation on a Workstation-Cluster differs significantly from the Meiko since we cannot make use of the fast interconnection network but are using an Ethernet connection for communication. Therefore, executing the program on more than one processor increases the execution time significantly because of the communication overhead. We have also tested the program using a process based implementation instead of the thread based one. As can be easily seen, a process based approach introduces a significant overhead if more than one SDA is mapped onto the same processor. This is due to the inability of explicitly scheduling processes. The situation becomes better if we enlarge the number of processors used, so that eventually only one SDA (and consequently only one process) is mapped onto a processor. However, the runtime on three processors is still worse than the sequential one because of the communication overhead. The execution time of the pthreads version is slightly higher on three processors - this is due to the context switching overhead between the execution and server thread; the process based implementation does not introduce this overhead.

Figure 12 presents the results for an extended producer/consumer example on the Meiko. Here, we have executed larger tasks on different numbers of processors where the whole produce and consume task is distributed over differing numbers of SDAs reaching from one to 16 producers and 16 consumers, respectively. Note, that due to the significant increase in the problem size (which was necessary to execute the program on up to 32 processors) the execution times cannot be directly compared with those given in Figure 11.

Again, it can be seen that the Opus implementation adds only limited overhead to the sequential version and that the problem scales well up to 32 processors. Moreover, having multiple SDAs executing on the same processor (as is the case if the number of producers and consumers is larger than the available processors) does not add a significant overhead and can indeed be advantageous because of a better overlapping of communication and computation.

*** Figure 12: Extended Consumer Producer Problem ***

7 Conclusion and Future Work

In this paper, we have presented our major design decisions for the Opus system and showed how it can be efficiently implemented. We have also identified low level services which have to be available on any target platform. In particular, these services include standardized packages for both communication and multithreading (such as MPI and pthreads respectively). The use of standardized packages enhances the flexibility and portability of the Opus system.

We have described in detail the runtime behavior of an Opus application and provided a specification for the supporting runtime system as well as for the restructuring compiler.

Our approach makes intensive use of a restructuring compiler which generates an HPF-conforming code expressing the Opus semantics. Thus, our approach is very different from runtime based approaches like KeLP [9] or HPF/MPI [14] where a runtime library is used to coordinate different tasks. By using a restructuring compiler our implementation is open for future extensions or modifications of the Opus language that might need more advanced compiler techniques (for example, the exploitation of intra-SDA parallelism). Another benefit of the compiler approach is that we do not need to tackle the interoperability problems of

Fortran 90 and C or C++ since all of our resulting code and most of our runtime system is written in Fortran 90.

Future work will focus on the implementation of the Opus compiler which will be realized in close relationship with the HPF compiler VFC [3]. Moreover the Opus runtime system has to be integrated with the VFC runtime system.

Acknowledgments

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. Matthew Haines was supported in part by NSF CAREER Award ASC-9623904.

References

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] S. J. Fink, S. B. Baden, and S. R. Kohn. Flexible Communication Mechanisms for Dynamic Structured Applications. In *IRREGULAR '96*, 1996.

[10] I. Foster and K.M. Chandy. Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, Vol. 26, 1995.

[11] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing Multiple Communication Methods in High-Performance Networked Computing Systems. *Journal of Parallel and Distributed Computing*, 40(1), Jan. 1997.

[12] I. Foster, C. Kesselmann, and S. Tuecke. Nexus: Runtime Support for Task-Parallel Programming Languages. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[13] I. Foster, C. Kesselmann, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel Computing*, 37:70–82, 1996.

[14] I. Foster, D.R. Kohr Jr., R. Krishnaiyer, and A. Choudhary. Double Standards: Bringing Task Parallelism to HPF via the Message Passing Interface. In *Proc. Supercomputing 96*, Pittsburgh, PA., 1996.

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

[16] W. Gropp and E. Lusk. *User's Guid for mpich, a Portable Implementation of MPI.* Argonne National Laboratory, 1996.

[17] T. Gross, D. O'Hallaron, and J. Subhlok. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel & Distributed Technology*, 2(3):16–26, Fall 1994.

[18] M. Haines. An Open Implementation Analysis and Design of Lightweight Threads. In *Proceedings of OOPSLA 97*, Atlanta, GA, October 1997.

[19] M. Haines, D. Cronk, and P. Mehrotra. On the Design of Chant: A Talking Threads Package. In *Supercomputing '94*, Washington DC, November 1994.

[20] M. Haines, B. Hess, P. Mehrotra, J. Van Rosendale, and H. Zima. Runtime Support for Data Parallel Tasks. In *Proceedings of The Fifth Symposium on the Frontiers of Massively Parallel Computation*, McLean VA, February 1995.

[21] 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.

[22] S.B. Hassen and H.E. Bal. Integrating Task and Data Parallelism Using Shared Objects. In *10th ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996.

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

[24] C. Hunt. *TCP/IP Network Administration.* O'Reilly and Associates, Inc., 1997.

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

[26] ISO. Fortran 90 Standard. ISO/IEC 1539 :1991 (E).

[27] LAM/MPI Parallel Computing. http://www.mpi.nd.edu/lam/.

[28] P. Mehrotra, J. Van Rosendale, and H. Zima. Language Support for Multidisciplinary Applications. *IEEE Computational Science & Engineering*, 5(2):64–75, 1998.

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

[30] 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.

```
program producer_consumer

  sda type buffer_type(size)
    integer :: size
    real, private :: fifo(0:size-1)
    integer, private :: count = 0
    integer, private :: px = 0
    integer, private :: 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 put
    subroutine get(x) when (count .gt. 0)
      real, intent(out) :: x
      x = fifo(cx)
      cx = mod(cx+1,size)
      count = count-1
    end subroutine get
  end buffer_type
```

```
sda type producer_type
  contains
  subroutine produce(b)
    sda(buffer_type) b
    real a
    do
      a = ... !  produce a
      call b%put(a)
    end do
  end subroutine produce
end producer_type

sda type consumer_type
  contains
  subroutine consume(b)
    sda (buffer_type) b
    real a
    do
      call b%get(a)
       ... = ... a ... ! consume a
    end do
  end subroutine consume
end consumer_type
```

```
! main program
  integer,parameter :: np=5
  integer,parameter :: nc=5
  integer,parameter :: buffersize=10
  sda(buffer_type) buffer
  sda(consumer_type) consumer(nc)
  sda(producer_type) producer(np)
  call buffer%create(buffersize)
  do i = 1,nc
    call consumer(i)%create
  end do
  do i = 1,np
    call producer(i)%create
  end do
  do i = 1,np
   spawn producer(i)%produce(buffer)
  end do
  do i = 1,nc
   spawn consumer(i)%consume(buffer)
  end do
end program producer_consumer
```

Figure 1: Producer/Consumer Problem with Opus

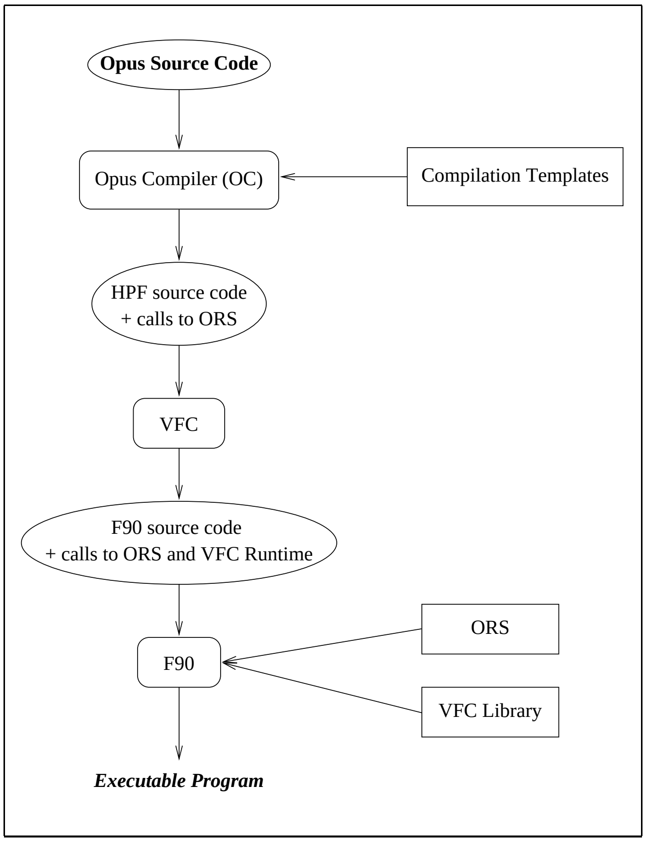


Figure 2: The Opus Compilation Process

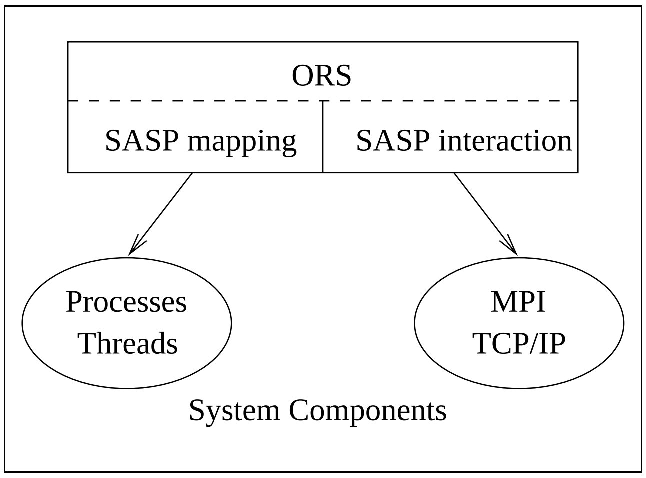


Figure 3: The Opus Runtime System and its Supporting System Components

```
TYPE ors_execution_rec
   INTEGER :: id, method_id, action, execution_type
   LOGICAL :: finished
   TYPE(ors_sda_handle), POINTER :: caller
   TYPE(ors_sda_handle), POINTER :: callee
   TYPE(ors_generic_type), POINTER :: arguments
END TYPE ors_execution_rec
```

Figure 4: The Execution Record

```
ors_initialize_sda(iq,oq)   [TYPE(ors_mi_queue) iq, oq]
ors_terminate_sda(oq)       [TYPE(ors_mi_queue) oq]

ors_get_rec(r, queue) result(return_rec)
                            [TYPE(ors_execution_rec), optional :: r
                             TYPE(ors_execution_rec), pointer :: return_rec
                             TYPE(ors_mi_queue) queue]
ors_insert_rec(r, queue)  [TYPE(ors_execution_rec) r
                             TYPE(ors_mi_queue) queue]
ors_delete_rec(r, queue)  [TYPE(ors_execution_rec) r
                             TYPE(ors_mi_queue) queue]

ors_new_event(e,r)          [TYPE(ors_event) e; TYPE(ors_execution_rec) r]
ors_test(e,s) result(finished)
                            [LOGICAL finished; TYPE(ors_event) e
                             TYPE(ors_sda_handle) s]
ors_wait(e,s)               [TYPE(ors_event) e; TYPE(ors_sda_handle) s]
ors_handle_inquiry(r,c,a)   [TYPE(ors_execution_rec) r,c; INTEGER a]

ors_handle_return(r)        [TYPE(ors_execution_rec) r]

ors_receive_mi(r)           [TYPE(ors_execution_rec) r]
ors_send_synchronous_mi(r)  [TYPE(ors_execution_rec) r]
ors_send_asynchronous_mi(r) [TYPE(ors_execution_rec) r]
```

Figure 5: Main ORS Routines

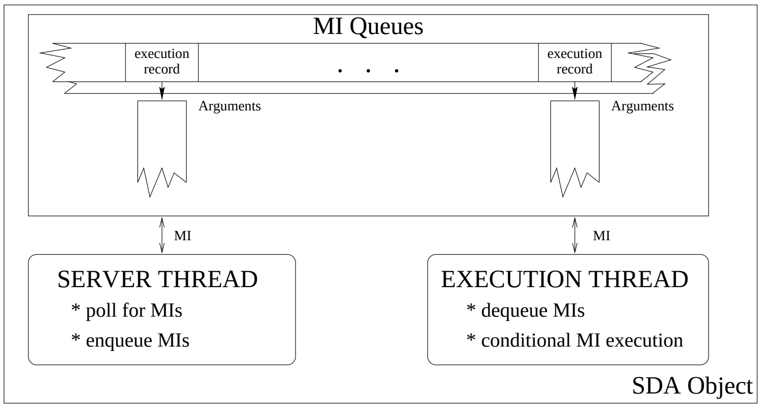


Figure 6: Structure of an SDA Object

```
SUBROUTINE method_1(b,...)
  TYPE(ors_sda_type), POINTER :: b
  TYPE(ors_execution_rec), POINTER :: r
  TYPE(ors_event) :: e
  ...
   ! CALL b%m1(...)
  r = ors_new_rec(b,method_id(b%m1), arguments)
  CALL ors_insert_rec(r,outqueue)
  CALL ors_send_synchronous_mi(r)
  ...
   ! e = SPAWN b%m2(...)
  r = ors_new_rec(b,method_id(b%m2), arguments)
  CALL ors_insert_rec(r,outqueue)
  CALL ors_new_event(e,r)
  CALL ors_send_asynchronous_mi(r)
  ...
   ! wait(e)
  CALL ors_wait(e,ors_my_sda)
  ...
END SUBROUTINE method_1
```

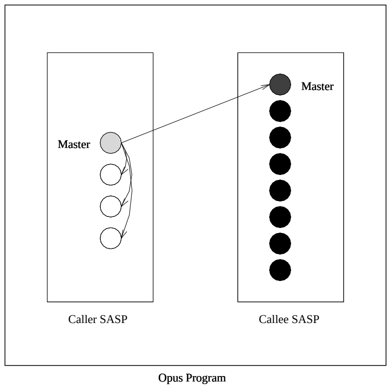
Figure 7: Compilation Template for Method Invocations

```
SUBROUTINE sda_name_server
  TYPE(ors_execution_rec), POINTER :: b,c
  TYPE(ors_mi_queue), POINTER :: inqueue
  TYPE(ors_mi_queue), POINTER :: outqueue

  DO  ! forever
    CALL ors_receive_mi(b)  ! blocking

    SELECT CASE (b%action)
    CASE (wait_id, test_id)
      c = ors_get_rec(b,outqueue)
      CALL ors_handle_inquiry(c,b,b%action)

    CASE (exec_ackn_id)
      c = ors_get_rec(b,outqueue)
      CALL ors_handle_return(c)

    CASE (method_call_id)
      SELECT CASE (b%method_id)
      CASE(1)
         ! receive arguments for method 1
        ...
      CASE(n)
         ! receive arguments for method n
      END SELECT
      CALL ors_insert_rec(b,inqueue)

    CASE (termination_id)
      EXIT
    END SELECT
  END DO
END SUBROUTINE sda_name_server
```

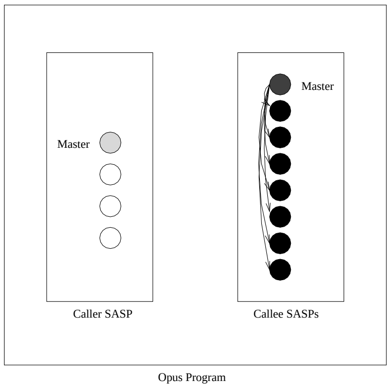
Figure 8: Compilation Template for the Server Thread

```
SUBROUTINE sda_name
  TYPE(ors_mi_queue), POINTER :: inqueue
  TYPE(ors_mi_queue), POINTER :: outqueue
  TYPE(ors_execution_rec), POINTER :: exec_rec
  ...
   ! user data
  ...
  CALL ors_initialize_sda(inqueue,outqueue)
  execution_loop: DO
    exec_rec = ors_get_rec(inqueue)
    SELECT CASE(exec_rec%method_id)
    CASE (1)
      IF (method_1_condition()) THEN
        CALL method_1(exec_rec%arguments ...)
         ! send results
        CALL ors_delete_record(exec_rec,inqueue)
      END IF
    ...
    CASE (n)
      ...
    CASE (termination_request)
      CALL ors_terminate_sda(outqueue)
      EXIT execution_loop
    END SELECT
  END DO execution_loop
CONTAINS
   ! method procedures and "when-condition" functions
END SUBROUTINE sda_name
```

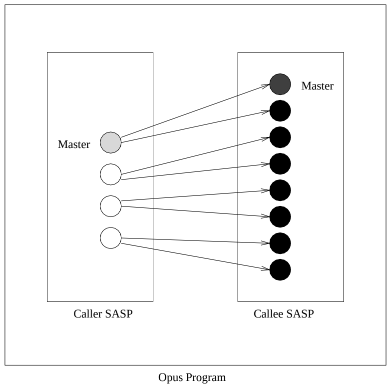
Figure 9: Compilation Template for the Execution Thread



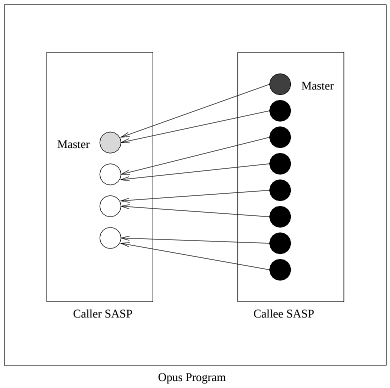
1. Caller master sends an MI to callee master with actual argument distributions and notifies its workers.



2. Callee master notifies its workers. All components of the caller and callee compute communication schedules.

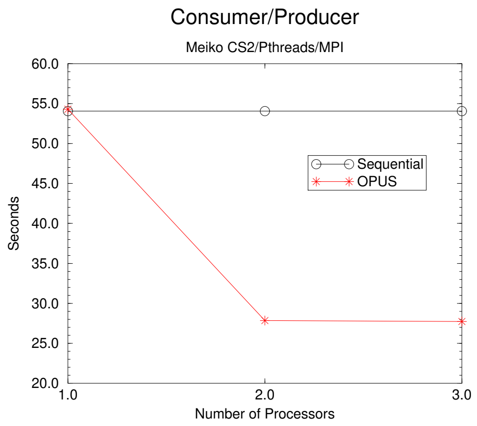


3. Caller components send data messages to appropriate callee components directly.

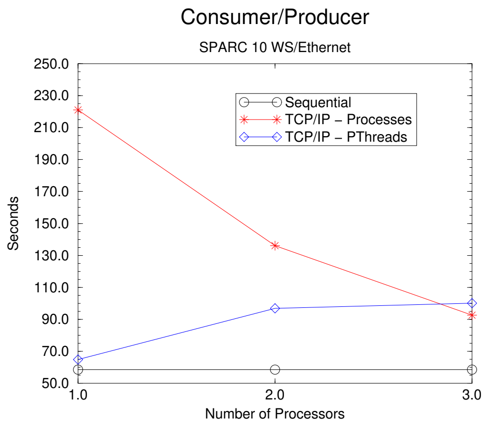


4. When method execution has finished, the callee components send any return messages to the caller components. This completes the MI.

Figure 10: Illustration of the MI Process for Distributed SDAs



(a)



(b)

Figure 11: Preliminary Results for the Consumer Producer Problem

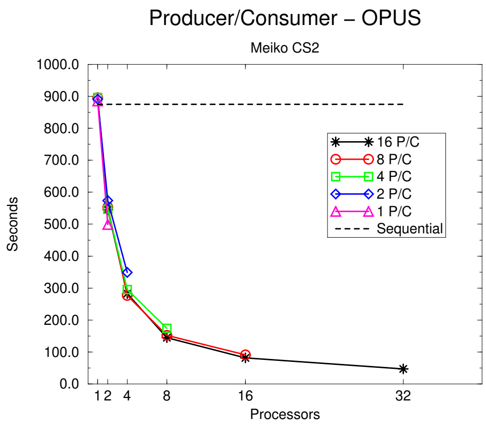


Figure 12: Extended Consumer Producer Problem