High Level Support for Distributed High Performance Computing

eingereicht von:

Erwin Laure

Dissertation

zur Erlangung des akademischen Grades
Doctor rerum socialium oeconomicarumque
(Dr. rer. soc. oec.)
Doktor der Sozial- und Wirtschaftswissenschaften

Fakultät für Wirtschaftswissenschaften und Informatik
Universität Wien

Erstgutachter: o. Univ.-Prof. Dr. Hans Zima
Zweitgutachter: Ao. Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta

Wien, im Februar 2001

Acknowledgments

I would like to thank my advisor, Professor Hans Zima, for his support and guidance on my thesis and for the opportunity to work in his excellent research group at the University of Vienna. Further I am grateful to my co-advisor Erich Schikuta.

Throughout the course of my research many people have contributed to this work. I am grateful to Piyush Mehrotra and Matthew Haines with whom I had numerous discussions on the initial design of Opus as well as on various implementation strategies. I would like to thank Siegfried Benkner for valuable discussions on HPF and for the support he gave me on the HPF compiler VFC. Many thanks to all the people from the VFC research group, in particular Bob Velkov, for their support on the internals of VFC. Further I am indebted to Bernd Wender for fruitful discussions on Java and the OpusJava design. I would like to thank Hans Moritsch for providing the stochastic linear programming example.

Finally, I am grateful to Siegfried Benkner, Thomas Fahringer, and Ingrid Ertl for partially proofreading this work and their helpful comments.

Abstract

Recent trends in hardware, in particular in interconnection technologies, have paved the way to the exploitation of heterogeneous, distributed computing platforms for advanced scientific applications. This infrastructure enables the building of *meta-applications* that are composed of several modules which may be implemented in different languages, exploit heterogeneous platforms, and employ several forms of parallelism. However, software development has not kept pace with the advances in hardware. Despite the existence of high level parallel languages and meta-computing frameworks there is, for the time being, a lack of a unified framework that efficiently deals with heterogeneity, interoperability, and hybrid parallelism at a high level.

In this thesis we develop a high level object based component system called **Opus** which allows the exploitation of hybrid parallelism in a multi-lingual and heterogeneous context. Opus is specifically targeted towards *Distributed High Performance Computing (DHPC)* where several (high performance) platforms are combined within a distributed environment. The main feature of Opus is an abstract data type, called *ShareD Abstraction (SDA)*, which encapsulates data and methods acting on this data and provides a clean interface for pasting together and coordinating different modules of a DHPC application. These modules may be implemented in any language for which an Opus binding has been defined.

The Opus system as we define in this thesis consists of two main components: the *Coordination Language Opus* which is a task parallel extension to High Performance Fortran, and *OpusJava* a Java based framework that is seamlessly interoperable with the coordination language. Specifically, the Opus language provides means for an efficient coordination of data parallel HPF tasks which may be dynamically instantiated on user specified components of a distributed system. Thereby, Opus provides an integrated hybrid parallel model which is indispensable for DHPC application. In this thesis we refine an older proposal for Opus and present advanced features for object migration, efficient object composition, priority scheduling, and intra-object parallelism, each of which is essential for DHPC applications.

While the Opus language is defined as an extension to HPF, OpusJava is a pure Java framework that introduces to Java the SDA concept together with the associated task parallel execution and coordination means. With the help of OpusJava any Java object may be dynamically instantiated on user specified resources and cooperate with any other object in the system. Additionally, OpusJava facilitates multi-lingual applications that are comprised of HPF and Java components.

Having specified the language extensions as well as the Java framework, we present a design of an environment that allows an efficient implementation of the Opus constructs on a wide range of architectures. We discuss a detailed execution model and derive associated runtime and compilation techniques. Interoperability issues among the two base languages (HPF and Java) are discussed as well. Specifically, we present generic interfaces and wrappers that allow an efficient interaction among these two languages. The internal structure of the OpusJava framework is also presented in detail.

To validate our techniques we have implemented prototypes of the Opus runtime system as well as of the OpusJava framework on a broad range of machines including the Cray T3E, the Quadrics CS2, and clusters of workstations. A prototype compiler that accepts a subset of the Opus language has been implemented by extending an existing HPF compiler (VFC). This compiler also generates the wrappers necessary for the interface between HPF and Java.

Contents

List of Figures

Chapter 1

Introduction

1.1 Distributed High Performance Computing

1.1.1 Applications

The past few years have dramatically changed the view of high performance applications and computing. While traditionally applications have been targeted towards dedicated parallel machines, we see the emerging trend of building "meta-applications" that are composed of several modules which may exploit heterogeneous platforms and employ several forms of parallelism. Examples of such applications include Multidisciplinary Applications, Problem Solving Environments, and Distributed Supercomputing Applications.

Multidisciplinary Applications. The high computing power provided by current high performance systems allows scientists and engineers to shift from single physical-model design to the design of a whole physical system with a large number of interacting models. These models, each of which represents a different discipline, are combined into a large multidisciplinary application (MDA) [138]. Examples include aircraft design, which contains models for an aircraft's structural and aerodynamic properties, and weather prediction where several submodels for clouds, wind, land, and sea have to work together.

Figure 1.1 depicts an environmental simulation of southern Florida [138]. The simulation comprises four distinct models representing different aspects of the Florida ecosystem. Each model is fairly complex and composed of various submodels. For example, the mesoscale-climate model comprises two submodels, one for wind and precipitation and another for cloud cover. The wind and precipitation model directly influences the swamp model, whereas the cloud cover model is only indirectly affecting the swamp model via the solar-radiation model.

Challenges in MDAs include the efficient coupling and coordination of the application's modules, the extraction and exploitation of parallelism within and across modules, as well as the integration of several languages and legacy code. A particular problem is the management of data transfer among modules. MDAs require not only efficient communication mechanisms for large amounts of data but also complex value interpolation and conversions between different data organization schemes.

Problem Solving Environments. Problem Solving Environments (PSEs) [86, 111] are specialized computing systems for solving a specific target class of problems. Specifically, PSEs contain highly tuned software systems, such as numerical libraries that are installed on high-end machines and may be accessed remotely in a client-server style. This access is either done explicitly or transparently by means of some middleware system. A prominent example of a numerical PSE is the NetSolve system which we describe in Chapter 7.

The main challenges in PSEs are how to implement the services provided in a highly efficient manner, tailored to the capabilities of the target high-end system, and how to make them accessible in an efficient, scalable, and secure way. Specifically, PSEs have to serve multiple clients

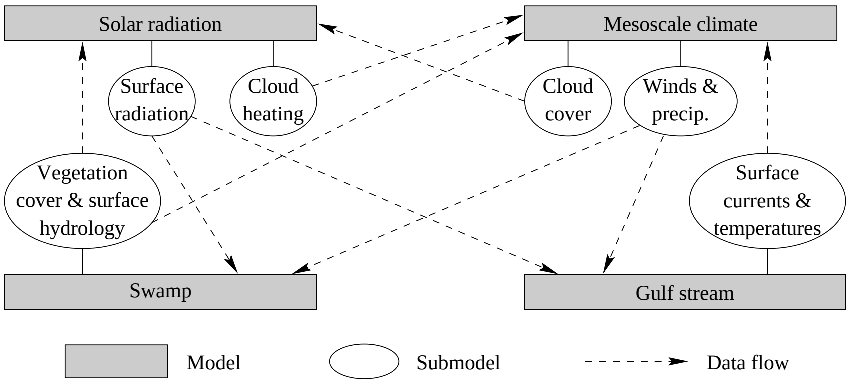


Figure 1.1: Models for Environmental Simulation of the Florida Swamp Ecosystem [138]

concurrently, they have to deal with security issues such as user authentication, and, most important, efficient data transfer from remote clients to the PSE has to be provided since PSEs may receive large amounts of input data.

Distributed Supercomputing Application. Distributed Supercomputing Applications are a class of applications whose computational requirements are so demanding that they can be met only by combining multiple high-end resources [142]. Distributed supercomputing applications are typically data intensive, complex simulations such as the analysis of astronomy data, on-demand processing of radar data, or computational chemistry applications.

The technical challenges of distributed supercomputing applications are similar to those of MDAs. In particular, appropriate techniques for composition and mapping of an application's modules are required in order to minimize data transfer and to provide good load balancing.

Summary. In all the application classes described some common properties and requirements can be identified:

- High compute power is of paramount importance.
- Often even multiple machines are needed to meet the application's requirements.
- The applications are composed of several modules that
  - exploit different forms of parallelism,
  - employ different data organization schemes and algorithms, and
  - are written in different languages.
- Legacy software needs to be integrated with newly developed code.

The requirements of the application classes discussed are challenging with respect to hardware as well as software development. First, their demand for computational power can only be met by high-end high performance machines. Second, their modular organization allows on the one hand the exploitation of loosely integrated architectures but requires on the other hand advanced programming support for an efficient coordination of the different modules in homogeneous as well as heterogeneous environments. Finally, programming models that leverage different, even hybrid, forms of parallelism are crucial for an efficient realization of these applications.

In the following, we refer to these classes of applications as **Distributed High Performance Computing (DHPC)** applications since they exhibit properties of both, classical high performance computing in that massive parallelism needs to be exploited, and distributed computing in that different modules of an application may be instantiated on different components of loosely integrated architectures. In the remainder of this section we give a brief introduction on the hardware systems employed for DHPC as well as the corresponding programming environments.

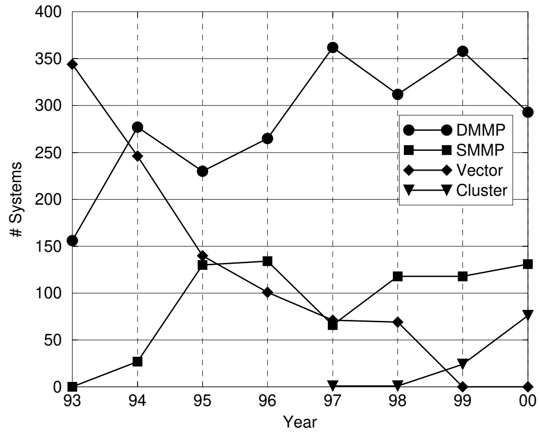


Figure 1.2: Architecture Share in Top500 List

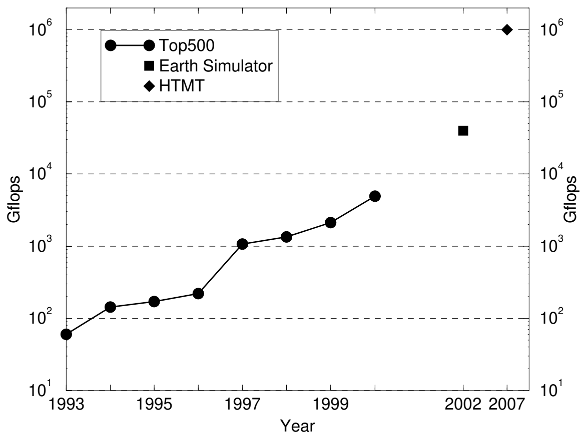


Figure 1.3: Supercomputer Performance

Source: Top500 List (http://www.top500.org),
ESRDC Earth Simulator Homepage (http://www.gaia.jaeri.go.jp/),
HTMT Overview (http://htmt.cacr.caltech.edu/Overview.html)

1.1.2 Computing Architectures for DHPC

DHPC applications are primarily targeted towards high end parallel machines because of their high computational requirements. A wide range of parallel architectures has evolved over the last decades and we can roughly distinguish between four major classes of parallel machines: *vector machines*, *shared memory multi-processors (SMMPs)*, *distributed memory multi-processors (DMMPs)*, and *hybrid systems* or *clusters*.

State of the Art Parallel Architectures. Figure 1.2 shows the share of any of these architectures in the list of the top 500 supercomputers [180] from 1993 to 2000. The two dominant architecture classes are currently SMMPs and DMMPs. The key property of SMMPs is that they provide a globally addressable memory and processors exchange information as a result of conventional memory access instructions. SMMPs can be further divided into two subclasses: *Symmetric Multi-Processors (SMPs)* and *NUMA Machines*. In SMPs all memory locations are equidistant to all processors, so all processors experience the same access time on a memory reference. Recent examples of SMPs include the SUN Enterprise and the (single-node) NEC SX-5 which uses vector processors as building blocks. However, the scalability of SMPs is limited due to technical reasons; current systems such as the SUN Enterprise are scalable up to 64 processors. *Non Uniform Memory Access (NUMA)* technology helps in overcoming the scalability problems of SMPs. In NUMA machines processors and memory modules are closely integrated such that access to local memory is faster than accesses to remote memories. Typical NUMA machines are for instance the SGI Origin or the HP Exemplar. In order to cope with the different memory access latencies in NUMA machines, multithreaded architectures that hide the latency of remote memory accesses by multithreading techniques have been proposed. A first experimental platform of this type is the Tera MTA [4] that provides up to 128 concurrent threads of control per processor, one of which is chosen for executing an instruction in each cycle.

DMMPs are machines that do not have a globally addressable memory. As a consequence, communication between processors has to be established via message passing rather than implicitly via conventional memory access instructions. This kind of architecture is virtually unlimited scalable. Current systems are comprised of several thousands of processors; for instance, one of today's most powerful machines, the ASCI Red with a peak performance of more than 3 Teraflops has 9,632 processors. Examples of other DMMPs include the NEC Cenju, the IBM SP-2, and the Quadrics CS-2.

Advances in standard interconnection technology allowed to move from highly integrated parallel machines to more loosely integrated clusters. This class of architectures often combines properties of both, SMMPs and DMMPs in that SMMPs are employed as basic building blocks of loosely integrated DMMPs. Because of the hybrid (shared and distributed) memory organization these systems are often referred to as *Hybrid systems.* Today's most powerful machine, the ASCI White with a peak performance of more than 12 Teraflops and a sustained performance of almost 5 Teraflops, can be classified as such a hybrid system since its 8,192 processors are organized within 512 distributed memory nodes, each node being a 16 processor SMP. With the advent of multi-processor PC boards and the Beowulf technology [163] this class of architectures is gaining more and more importance as reflected in Figure 1.2. For a more comprehensive discussion see for instance [62].

Performance Prospects of High-End Machines. Figure 1.3 shows the sustained performance (measured using the LINPACK benchmark) of the most powerful machines from 1993 to 2000. Currently the most powerful machines are in the range of a few Teraflops but major advances are expected in the near future. For instance, the goal of Japan's *Earth Simulator* [67] project is to build a 40 Teraflops machine by 2002. This machine will be based on clusters of vector SMPs. The HTMT project in the US [170] envisions to build the first Petaflops machine by 2007. The architecture designed for this machine no longer belongs to any of the classes discussed above but is comprised of advanced new technologies such as multithreaded architectures, Processor-in-Memory (PiM) smart memory (see for instance [90]), and optical storage. Another Petaflops effort is currently undertaken at IBM. IBM's Petaflops machine, named *Blue Gene* [32], will be comprised of 1 million processors, 32 of which on a single chip tightly integrated with the memory.

Metacomputers and Complex Clusters. High peak performance in the Teraflops range is for the time being only provided by a few, very cost intensive machines. A more cost effective—though viable—alternative to these high-end machines is the exploitation of *existing* machines in a cooperative way. Smarr and Catlett envision a transparent exploitation of computational resources within a national or even world-wide network. They call such a setting a *metacomputer*: *"The metacomputer is a network of heterogeneous, computational resources linked by software in such a way that they can be used as easily as a personal computer."* [167]

These ideas have been taken up and refined by Foster and Kesselman who envision a *computational grid* that is *"a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities."* [80]

Metacomputing systems have to deal with a number of organizational and technical issues not present in high-end machines. Apart from the inherent hardware heterogeneity of metacomputers, metacomputing systems have to cope with different administrative domains, security issues, and unpredictable bandwidth changes in wide area networks, to name only a few of the most important issues. A local collection of computational resources, such as the machines of a computing center, is a special kind of metacomputer that does not expose all of the problems stated above. We refer to such collections as **Complex Clusters**. More specifically, we define complex clusters as follows:

> *Complex Clusters are heterogeneous collections of well defined computational resources that are connected by a local area network, run under the same administrative domain, and can be used as one capable supercomputer.*

This definition comprises several properties which deserve a more in depth treatment:

Heterogeneous collections. Complex clusters can be collections of all kind of state-of-the-art machines, specifically collections of DMMPs, SMMPs and Hybrid Systems. Hence, complex clusters are typically characterized by heterogeneity in processor technology, interconnection technology, memory organization, and operating systems.

Well defined computational resources. As opposed to general metacomputers, the components of a complex cluster are well defined in terms of their capabilities. In other words, the user of a complex cluster knows a priori which machines form the system, what the power of any component is, and whether a component runs in single- or multi-user mode.

Local area network. The components of complex clusters are interconnected by special purpose local area networks. Hence, specific interconnection properties, in particular sustained bandwidth, can be guaranteed as opposed to collections of computers connected by the Internet.

One administrative domain. All components of complex clusters are managed under one administrative domain. This restriction greatly reduces problems such as security or accounting, which are of crucial importance in general metacomputing systems.

Complex clusters are a practical and cost effective alternative to high-end systems in the field of DHPC. In particular, the combination of properties from parallel, high performance architectures being the building blocks of complex clusters, and distributed computing makes complex clusters well suited for DHPC applications. In the remainder of our discussion we consider complex clusters as the main target architecture.

1.1.3 Programming Environments for DHPC

Program development for parallel and distributed architectures is much more complex than for sequential machines. The programmer not only has to ensure algorithmic correctness but also to deal with all the details of parallelism, specifically with the *decomposition* of a program into concurrently executing parts, the *mapping* of these parts to machines and the processors of parallel machines, the exchange of information (*communication*), and the *synchronization* of the components that execute in parallel.

Low Level Programming Models. At a low level the programmer is provided with machine specific parallel programming models that typically coincide with the physical organization of a parallel architecture. Specifically, SMMPs provide the *shared memory model*, where a program consists of one or more *threads of control*, each operating in an address space that contains a region shared between threads, and may contain a region that is private to each thread. The user has to split the program into appropriate threads which have to be appropriately synchronized. DMMPs provide the *message passing model* which assumes a collection of *processes* each operating in a private address space and each able to communicate with other processes. Processes communicate with each other through the exchange of messages which are inserted in the code by the programmer. SMMPs are also often programmed under the message passing paradigm because it allows to express data locality, which is of particular importance for NUMA machines. A popular methodology for implementing message passing programs is the *single program multiple data* (SPMD) model. Within this model all processes execute the same program but on different parts of the data space.

However, writing efficient parallel programs by using these low level models is not an easy task. The message passing programming style forces the user to explicitly manage all details of data distribution, work distribution, and communication whereas the shared memory programming style requires proper thread synchronization to ensure data consistency. Programming under these models is therefore a tedious, time-consuming, and error prone task, which can be compared to assembly programming on a sequential machine.

High Level Programming Models. As a consequence of the problems involved with low level parallel models a large number of parallel languages and systems that aim to relieve the user from the burden of tackling all details of parallelism explicitly has been developed. High level support is primarily provided for two popular parallel programming models: the *data parallel* and the *task parallel* paradigm.

In the data parallel model a program is written assuming a single thread of control and a global address space. Parallel operations are applied concurrently to different parts of the data space. Because of the global address space communication is implicit and data can be accessed via variable references. The distribution of data structures onto a set of processes influences the distribution of parallel computations. Data parallelism is primarily provided by data parallel languages that extend standard languages such as Fortran and C with means for specifying data distribution and parallel code segments. Compiler and runtime technologies work together to map data parallel constructs to parallel machines. Examples of such languages include HPF [106] and its predecessors Vienna Fortran [58] and Fortran-D [108], HPC++ [88], and HPJava [42]. Data parallel programs scale as long as the problem size is large enough; the performance of programs with small problem sizes may however degrade if a large number of processors is used. This is mainly due to synchronization and communication overheads.

Task parallel programs consist of a set of potentially dissimilar tasks that communicate and synchronize explicitly. Task parallelism provides more flexibility than data parallelism, however the user is much more involved in the details of parallelism. Moreover, it is often difficult to split a problem into appropriate tasks such that the target platform is exploited efficiently. Specifically, a large number of processors requires a large number of tasks which is often not available within an application. Task parallelism is provided by task parallel languages such as Linda [45], Fortran M [75], Fx [96], CC++ [52], and Orca [11]. With the growing popularity of distributed object computing a number of middleware [28] systems that provide task parallel features have been developed. Examples include CORBA [155] and Microsoft's DCOM [110].

Traditionally, either data parallelism or task parallelism has been employed in parallel programs. However, it was soon realized that many applications need to harness both forms of parallelism, often even in an interleaved way. With the advent of hybrid cluster systems the need for intermixing both paradigms became even more evident: the programmer requires means that allow to obey the different memory access characteristics of such architectures in order to exploit the potential power of the machine best. Consequently, data parallel languages have been extended with task parallel features and vice versa, such as the task parallel extensions of HPF-2 [107] and the data parallel extensions of Orca [18]. A more detailed discussion of related parallel systems is given in Chapter 7. A comparative overview of languages integrating task and data parallelism can be found in [10].

Requirements for DHPC Applications. An efficient exploitation of hybrid forms of parallelism is of paramount importance for DHPC applications. Additionally, DHPC applications require support for heterogeneous platforms as well as heterogeneous programming environments, i.e. support for combining different languages and supporting libraries within a unified framework. Specifically, we can identify three major criteria for programming systems for DHPC applications:

Heterogeneity: Programming environments for DHPC applications have to cope with the different characteristics of hardware components in a heterogeneous environment. This involves in particular abstractions from different operating systems and differences in data representation. The mapping of tasks onto the components of a complex cluster also deserves particular attention. Specifically, dynamic means for establishing the mapping among tasks and hardware components, including dynamic task re-location, are required.

Interoperability: Apart from heterogeneity in hardware, DHPC applications are often heterogeneous in software, too. This kind of heterogeneity is often referred to as *software interoperability*.

One of the most important interoperability issues is *language interoperability*. Making different languages interoperable not only involves calling interoperability (i.e. the possibility of invoking procedures written in a particular language from within another language) but also interoperability of language specific runtime and communication systems. Moreover, different data and storage formats need to be obeyed. An important aspect of language interoperability is the possibility of integrating legacy code into DHPC applications. In the following, we refer to this aspect of interoperability as *horizontal interoperability* since it tackles issues that occur primarily among systems at the same level of abstraction.

In addition to language interoperability, DHPC applications often require the integration of highly optimized supporting systems, such as numerical libraries or specific communication systems available on a particular machine. In this context escape mechanisms that allow to leave a higher level model, such as the data parallel model, and implement parts of an application directly in a lower level model, such as message passing, are essential. Since this aspect of interoperability deals with systems at different levels of abstraction we introduce the term *vertical interoperability.*

Parallelism and Communication: Finally, DHPC applications need to harness different kinds of parallelism, in particular data (and fine grained task) parallelism within components of an application and task parallelism across components. Flexible means for interaction among concurrently executing components are required as well. All issues related to parallelism, i.e. program decomposition, mapping, communication, and synchronization should be tackled at the highest possible level in order to ease program development and maintenance.

Apart from these criteria, DHPC applications may have to deal with several other issues such as fault tolerance, security, or load balancing in multi-user environments. However, since we assume complex clusters as the target architecture for DHPC applications, the importance of these issues is greatly reduced.

In the context of metacomputing several systems have been developed that provide support for (most) of these issues. Representative examples include Globus [78], WebFlow [104], and Legion [93]. However, these systems are either at a low level that is comparable with message passing or do not sufficiently support hybrid forms of parallelism (see Chapter 7 for a more detailed discussion of these issues).

Because of the lack of a unified framework that efficiently deals with all the issues discussed, there is currently a tradeoff between high level coordination of application modules and efficient high level means for parallelism. A unified programming environment that deals with heterogeneity, interoperability, and hybrid parallelism at the highest possible level is therefore of paramount importance.

1.2 Thesis Outline

The goal of this thesis is to develop a high level component system for distributed high performance computing that provides support for the main criteria discussed above. This system, which we call **Opus**, provides high level abstractions for hybrid parallelism in a multi-lingual context. Specifically, Opus provides high level coordination means for data parallel components, bindings to languages as different as HPF and Java, and explicit support for heterogeneous architectures. The Opus component model is based on a construct called *ShareD Abstraction* (or *SDA*). SDAs encapsulate data and methods acting on this data and provide a clean interface for pasting together and coordinating different modules of a DHPC application.

In this thesis we introduce Opus by means of its bindings to High Performance Fortran (called *The Coordination Language Opus*) and Java (called *OpusJava*). The SDA construct, building the basis of Opus, provides interoperability among these two language bindings and, thereby, facilitates multi-lingual DHPC applications.

Specifically, we will define the *Coordination Language Opus*[1] as a task-parallel extension to High Performance Fortran (HPF). Opus provides means for coordinating concurrently executing HPF tasks which can be instantiated dynamically on specific components of a complex cluster. We split our discussion of the Opus language into two parts. Part 1 (*the basic language*) contains all the features that are necessary for a basic coordination of HPF tasks; we specifically deal with issues in integrating task and data parallelism at a high language level by refining an older proposal of Opus. Part 2 (*advanced features*) contains features that are of particular importance for DHPC applications such as dynamic task re-location, lightweight objects, and intra-object parallelism.

[1] In the following, when it is clear from the context, we will simply use the term *Opus* when referring to the Coordination Language.

Moreover, we develop a Java binding of Opus, called *OpusJava*, that provides an interface to the Coordination Language Opus. OpusJava facilitates a seamless integration of HPF and Java modules within DHPC applications. With help of OpusJava modules of either language are interoperable and can be coordinated in a high level manner. This allows, for instance, to make HPF applications accessible remotely via the World Wide Web. OpusJava may in addition be exploited as communication mechanism among heterogeneous components of a complex cluster because of Java's excellent support for heterogeneous architectures.

Apart from the language and framework definition we develop compilation and runtime techniques for an efficient execution of Opus programs on complex clusters. Specifically, we describe how the Coordination Language Opus is compiled to equivalent HPF components and which runtime support is needed for the coordination of these HPF components. Moreover, we derive interfaces and wrappers that allow a seamless and efficient integration of HPF and Java modules by means of the OpusJava framework.

To validate our techniques we have implemented a prototype Opus compiler that accepts a subset of the Coordination Language Opus, a prototype runtime system, as well as a prototype of the OpusJava framework. With this prototype system we conducted a number of experiments that validate the efficiency of the language and framework constructs introduced.

The thesis is organized as follows:

The basic Coordination Language Opus is introduced in Chapter 2. The language description is kept rather informal and most of the features are illustrated by means of examples. The formal syntax is developed in Appendix A; intrinsic Opus functions are summarized in Appendix B. This chapter covers the basic Opus features, in particular its support for encapsulating HPF programs within computational modules that are called *ShareD Abstractions (SDAs)*, and the basic coordination mechanisms. In presenting Opus we assume some familiarity with HPF. A brief introduction to the basic concepts of HPF can be found in Chapter 7. For a profound discussion see for instance [107, 122]. After introducing these features we validate them against the criteria presented in Section 1.1.

In Chapter 3 we define advanced Opus features that are designed to remove problems of Opus with respect to DHPC applications as identified in Chapter 2. In particular, we introduce features that allow the exploitation of more flexible and fine grained task parallelism as well as features that remove some of the overheads induced by the SDA mechanism. Moreover, dynamic task migration features which are indispensable for DHPC applications are developed.

The Java binding of Opus, called *OpusJava* is developed in Chapter 4. This binding also includes an interface to the Coordination Language Opus in order to make Java and HPF interoperable. Specifically, OpusJava is a Java component framework that allows a seamless integration of Java and HPF modules by means of the well defined SDA concept. We introduce the framework rather informally by means of examples. The detailed API can be found in Appendix C.

The design of an Opus implementation is presented in Chapter 5. We discuss the compilation of the Opus language constructs as well as the required runtime support. Moreover, we present the design of the OpusJava framework and discuss its interface to HPF.

In Chapter 6 we present a number of examples and experiments for both the Coordination Language Opus and OpusJava.

A detailed overview of related approaches to parallel and distributed computing as well as distributed high performance computing is given in Chapter 7. We develop a high level classification and present representative approaches in detail. After introducing the basic features we evaluate the approaches against the criteria presented in Section 1.1 and contrast them to our Opus approach.

Finally, in Chapter 8 we summarize the contributions of this work and conclude with a discussion of future research.

Chapter 2

The Coordination Language Opus

2.1 Introduction

The Coordination Language Opus (or simply Opus in the following) has been designed in a joint effort of ICASE, NASA Langley Research Center, and the Institute for Software Science to support the development of multidisciplinary applications (see [54, 55, 133, 138]). In this chapter we present the basic design of Opus, which we refine and augment in some minor aspects, in the context of multidisciplinary applications. Issues in the context of more general DHPC applications are sketched in Section 2.7 and elaborated in Chapter 3.

Multidisciplinary applications (MDAs), which are among the most demanding DHPC applications (cf. Section 1.1), are typically constructed by pasting together modules from a variety of related scientific disciplines. Examples of MDAs include environmental simulation, where several modules that model different parts of the environment (such as a sea model, a wind model, etc.) are combined, or modern aircraft design which requires a wide variety of interacting disciplines: aerodynamics, propulsion, structural analysis, controls, and so forth.

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

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

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

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

To fulfill these requirements, Opus introduces a new construct called a *ShareD Abstraction (SDA)*. This concept supports the development of MDA codes by providing data and method encapsulation. We use the well-developed HPF [107] facilities for data parallelism within each

SDA, while borrowing ideas from operating systems for inter-module communication and task-level parallelism.

Opus extends HPF/Fortran 90 in several areas, including *encapsulation*, *task parallel execution*, and *task coordination* features:

Encapsulation Features. The coordination of complex multidisciplinary codes, where separate modules might be written by different groups using different data structures and approaches, requires the possibility of encapsulating modules with a clean interface.

Opus provides the *ShareD Abstraction (SDA)* construct which encapsulates data and the methods manipulating the data. SDAs can be used as computation servers as well as shared data repositories. Although SDAs borrow heavily from the Fortran 90 module concept, they are conceptually data types and provide therefore more flexibility. We discuss SDAs in detail in Section 2.3.

Task Parallel Execution Features. An efficient implementation of multidisciplinary codes requires different levels of parallelism: while the data parallelism available within a module can be efficiently exploited by state of the art data parallel languages such as HPF, this kind of applications also requires coarse grained task parallelism between modules.

The task parallel features of Opus allow a parallel execution of SDA methods which may internally make use of data parallelism expressed through HPF directives. Thus Opus provides a programming model which is able to exploit multiple levels of (nested) parallelism. Section 2.4 describes the task parallel execution features of Opus.

Task Coordination Features. In order to make an efficient use of task parallel features the programmer must have the possibility of coordinating and synchronizing parallel tasks in a flexible and general way. Opus provides both implicit and explicit synchronization features which give the programmer a high degree of flexibility. These features are presented in Section 2.6.

The remainder of this chapter is organized as follows: We summarize the salient features of Opus in Section 2.2. A detailed treatment of the Opus language can be found in Sections 2.3 through 2.6. We introduce the syntax and semantics of Opus rather informally by developing an Opus solution to the standard producer-consumer problem[1]. A full Opus formulation of the producer-consumer problem can be found in Chapter 6. The formal syntax of Opus is developed in Appendix A; intrinsic Opus functions are summarized in Appendix B. In describing the Opus language, we assume some familiarity with HPF. Finally, we evaluate Opus by means of the criteria developed in Chapter 1 in Section 2.7.

2.2 Opus at a Glance

2.2.1 Language Features

Opus introduces a small set of extensions to HPF 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 from Fortran 90 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 and that the code of SDA methods may employ HPF parallelism 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*, which is bound to an *SDA variable*, 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

[1] This simple problem, in which a set of producers generates data that is processed by a set of consumers, is also the basis for a number of real-world applications. Our version creates a system in which each individual producer and consumer operates independently. Synchronization between them is provided by controlling their access to a bounded FIFO buffer.

SDA the resources on which it will execute are allocated, all the internal data of the SDA is also allocated and initialized in order to establish a well-defined initial state, and the object is bound to an SDA variable. Resource allocation on SDA creation 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 its associated SDA variable. The invocation of a method of an SDA creates a new *task*. SDAs represent distinct address spaces hence Opus tasks do not directly share data but may communicate by invoking methods in other SDAs. Method arguments are passed with copy-in/copy-out semantics. These quite restrictive features, which help to preserve a consistent state of the data of an SDA, will be extended in Section 3.3.

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 the status of a task without having to wait for its completion, or it may *wait* until a method execution has finished. No two methods of an SDA are allowed to execute in parallel; thus each method has *exclusive access* to the data of the SDA, much like in monitors [109]. A method may have an associated *condition clause* specifying a logical expression which guards the method's activation. Features that allow concurrent execution of multiple methods within an SDA will be introduced in Section 3.2.

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.

The execution of an Opus statement may be bound to a *status variable* which must be an allocated integer variable. It is set to zero if the execution of the associated statement was successful and it is set to a processor-dependent positive value if there is an error condition. If there is no status variable, the program terminates when an error condition occurs.

2.2.2 Execution Model

Each SDA object is associated with a unique *SDA context* which provides an address space for the SDA's data and operates one or more[2] *SDA threads* which are the locus of all control activity related to the SDA. Multiple SDA threads are called an *SDA thread pool*. An SDA context is established on the resources allocated to the SDA, called *nodes*. A node is a collection of one to several physical processors, each of which having access to the same (shared) memory. Although also processors of different nodes may have access to shared memory, we assume that in general such processors do not have direct access to the same memory. In our terminology, a node may for instance be a processor of a DMMP, a processor of an SMMP, or an SMMP within a cluster. An SDA context which is set up on multiple nodes is called a *distributed SDA context*. Distributed SDA contexts operate SPMD style, data parallel SDA threads that are equivalent to HPF threads. The execution of an Opus program can be thought of as a system of SDA threads in which a thread executes a method of its SDA in response to a request from another SDA. Figure 2.1 depicts these concepts: within an Opus program, two SDA contexts are established and two nodes are allocated to each of them. While the first context operates a single SDA thread (*t1*), the second one operates three of them (*t1* - *t3*). To exemplify the SPMD nature of SDA threads they are depicted in any node belonging to their SDA context.

An Opus program execution consists of a *main procedure* and a set of *SDA objects* which are created and destroyed dynamically during the lifetime of the main procedure. The main procedure is, like SDA objects, associated with a unique SDA context such that it has its own address space and SDA thread. In order to exploit the available resources efficiently, multiple SDA contexts may

[2] Multiple SDA threads allow a concurrent execution of multiple SDA methods as will be discussed in Chapter 3.

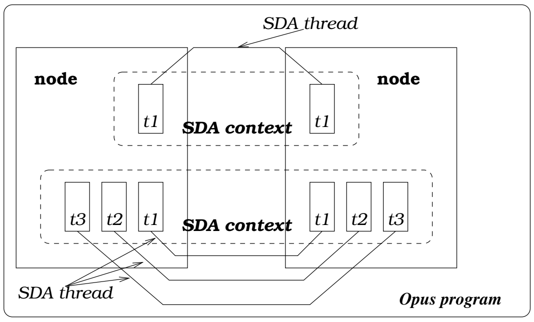


Figure 2.1: Opus Execution Model Components

be established on a node.

The execution of an Opus program starts by executing the main procedure, so that we begin with a single (possibly distributed) SDA context. When new SDAs are created, new SDA contexts are established on explicitly specified sets of nodes. All the SDA threads of an SDA context run asynchronously and in parallel with all other SDA threads in the program.

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 condition clause, one of these buffered requests is chosen to be fulfilled, that is, the associated method is executed. An SDA thread is free to chose any request with a satisfied condition clause for execution; Opus makes no guarantees with respect to fairness of the scheduling mechanism of an SDA thread. With the help of condition clauses a user may impose scheduling constraints, though.

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. Note that an SDA is able to receive method invocation requests without involving its associated SDA threads, i.e. independent of current state of their SDA threads.

An SDA thread may be in one of four different states:

1. *waiting* on requests to be fulfilled (the initial state),
2. *evaluating* condition clauses and selecting a request for being served,
3. *executing* a method, or
4. *blocked* in synchronization with other SDAs.

Figure 2.2 depicts the state transition diagram for an SDA thread. After its creation, the SDA thread is *waiting* for method invocation requests from other SDAs. Having received a method invocation request, the SDA thread is *evaluating* the condition clauses associated with the methods for which invocation requests are buffered. A request is said to be *ready for execution* (or simply *ready*) if the corresponding condition clause evaluates to **true** or no condition clause is specified for the respective method. From all the ready requests one is chosen for being served and the SDA thread starts *executing* the method associated with the request. If no ready requests are available the SDA thread becomes *waiting* again. During the execution of a method, the SDA thread may be *blocked* due to synchronous method invocations or to synchronization with event variables (waiting on an event). After the termination of the method execution the SDA thread starts evaluating the condition clauses of the buffered requests again.

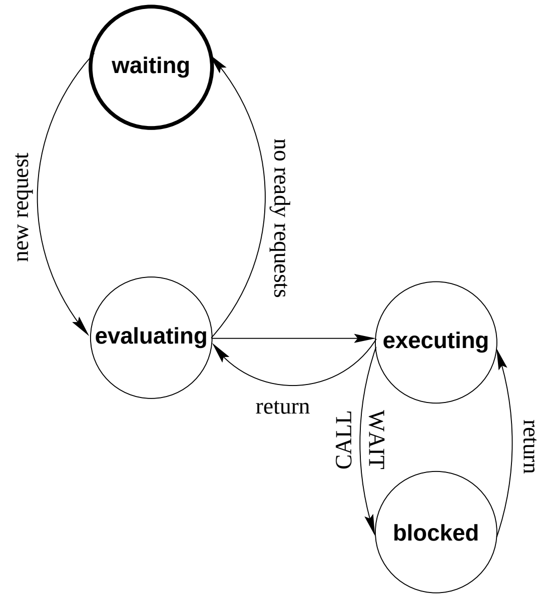


Figure 2.2: SDA Thread State Transition Diagram

2.3 ShareD Abstractions

2.3.1 SDA Types

The core of Opus is the *ShareD Abstraction (SDA)* construct. An *SDA type* specifies an object structure that contains data along with the methods (i.e. procedures) which manipulate this data. The syntax of the SDA type is modeled on the Fortran 90 module syntax and, in general, consists of the following components:

- a *name* and a list of *dummy arguments* which can be used to parameterize the SDA type,
- a *specification part* declaring the required types and data structures together with their distributions and other attributes, and
- a *method part* which specifies *methods* acting on the SDA's internal data structures.

The definition of an SDA represents a *parameterized SDA type.*

2.3.1.1 SDA Type Arguments and Specification Part

The dummy arguments of an SDA type specification can be used to parameterize the internal data structures of the SDA (including local arrays). They conform to Fortran 90 procedure conventions and must be of intent **IN**. When an SDA object is created (cf. Section 2.4.3), the dummy arguments are bound to actual arguments and a *fully specified* SDA type is obtained.

The specification part declares the internal data structure of an SDA. Its syntax is similar to the Fortran 90 module specification part. All entities declared inside the SDA type are considered public unless explicitly specified as **PRIVATE**. In addition to the Fortran 90 syntax the attribute **READ_ONLY** can be used to restrict the access to a public variable. As it is obvious from the name, **READ_ONLY** entities may be read by any SDA having access to them, but such an entity may only be modified by the SDA in which it is declared.

Public variables of an SDA are visible and accessible to all tasks having access to the SDA. They can be accessed using intrinsic method calls (cf. Section 2.4.3) and every Fortran 90 read or write access is implicitly transformed to such a method call. Thus, in order to avoid incoherent states of the SDA's data, all operations manipulating an SDA have exclusive access to its data and are executed as atomic operations.

The specification part may also contain HPF-directives for specifying processor arrays, data distribution, and alignment in order to enable data parallel executions of the methods.

Example 2.3.1: SDA Specification Part

```
      SDA TYPE buffer_type(size)
!HPF$ PROCESSORS directives
       INTEGER, PRIVATE :: size
       REAL, PRIVATE :: fifo(0:size-1)
       INTEGER, PRIVATE :: count=0
       INTEGER, PRIVATE :: px=0, gx=0
!HPF$ ALIGN & DISTRIBUTE directives
       . . .
      CONTAINS
       ! method part
      END SDA TYPE buffer_type
```

The code fragment above shows the SDA type definition for a bounded buffer where the buffer (`fifo`*) is realized as an array. The dummy argument* `size` *is used to parameterize the size of the buffer. HPF directives may be included into the specification part to allow data parallel computation.* ■

2.3.1.2 SDA Method Part

Methods are arbitrary procedures (either functions or subroutines) and may have arguments of any intent which are in general passed with copy-in/copy-out semantics.

A method may be associated with a *condition clause* which contains a side-effect free logical expression that controls the execution of the method in the sense that a call to the method can be executed only if the expression evaluates to **true**. If it is **false** the method invocation request is deferred until the condition evaluates to **true**.

Method Arguments. The dummy arguments of a method fall into three categories:

- Arguments conforming to Fortran 90 conventions:
  As mentioned above, these arguments may have an arbitrary intent (according to Fortran 90 conventions) and are passed with copy-in/copy-out semantics.
- SDA arguments:
  A dummy SDA argument represents a (possibly undefined) reference to an SDA object. It is transferred according to call-by-reference semantics.
- EVENT arguments:
  A dummy EVENT argument represents a (possibly undefined) reference to an event. It is transferred according to call-by-reference semantics.

Method dummy arguments may be distributed using the rules applicable to procedure arguments in HPF. Furthermore HPF parallel statements and directives may be used to explicitly express parallelism.

Example 2.3.2: SDA Type Definition

The full definition of the SDA type `buffer_type` *from Example 2.3.1 is given below. Here* `fifo` *is managed as a cyclic data structure implementing a FIFO discipline. Two* *methods,* `putb` *and* `getb` *are defined for accessing the buffer. Both methods are guarded by condition clauses which check for underflow and overflow of the buffer, respectively. Dummy arguments to these methods are conforming to Fortran 90 conventions.*

```
      SDA TYPE buffer_type(size)
!HPF$ PROCESSORS directives
       INTEGER, PRIVATE :: size
       REAL, PRIVATE :: fifo(0:size-1)
       INTEGER, PRIVATE :: count=0
       INTEGER, PRIVATE :: px=0, gx=0
!HPF$ ALIGN & DISTRIBUTE directives
       . . .
      CONTAINS
       SUBROUTINE putb(x) WHEN (count .LT. size)
         REAL, INTENT(IN) :: x
         fifo(px) = x
         px = MOD(px+1,size)
         count = count + 1
       END

       SUBROUTINE getb(x) WHEN (count .GT. 0)
          REAL, INTENT(OUT) :: x
          x = fifo(gx)
          gx = MOD(gx+1,size)
          count = count - 1
       END
       . . .
      END SDA TYPE buffer_type
```

■

2.3.2 SDA Declaration

An SDA declaration introduces one or more *SDA variables*. The declaration and all associated variables are called *typed*, if an SDA type name is specified in the declaration, otherwise *generic*. The syntax is similar to that of Fortran 90 type declaration statements referring to a derived type. Unlike standard Fortran variables, SDA variables are default initialized to be disassociated, i.e. not bound to an SDA object.

Assume that v is either an object declared in the entity declaration list, or, if the object is an array, an associated array element. Then v can be used to *create* an SDA object (cf. Section 2.4.3); after creation, v a associated and serves as a handle to that object.

For a generic declaration, v can be bound to an SDA object of arbitrary type. Otherwise, the specified parameterized SDA type and the object type must conform.

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

An SDA variable can be an argument for procedures and methods. An explicit interface is required for external procedures with SDA dummy arguments, similar to F90 derived types.

Generic SDA Variables. The creation of an SDA object in connection with a generic SDA variable is only possible by referring to an external SDA object (cf. Section 2.4.3): the new object inherits the type and state of the external object. Note that using this facility implies runtime checks to determine whether a method called with such an object exists and, if it does, whether the argument types match. However, a judicious use of status variables provides a graceful failure mode.

Example 2.3.3: SDA Declaration

The following code fragment declares SDA variables `S1`, `S2`, *and* `S3`, *all of which are associated with the parameterized SDA type* `buffer_type`. `S4` *is a generic SDA variable.*

```
SDA (buffer_type) :: S1
SDA (buffer_type) :: S2(n,m)
SDA (buffer_type), ALLOCATABLE, DIMENSION(:) :: S3
SDA S4
```

■

2.4 Method Activation

Public methods associated with an SDA may be invoked by tasks having access to that SDA. The successful activation of a method creates a task. Tasks are atomic in the sense that no two methods associated with the same SDA may be active at any given time. Syntactically, a method of an SDA is referenced by its method-name, separated from the SDA-variable-name by the SDA-separator "." which is used instead of the standard Fortran 90 separator "%".

Depending on the calling conventions and on the resulting relationship between caller and callee, two kinds of method activations are distinguished:

2.4.1 Synchronous Method Activation

A synchronous method activation blocks the caller until the callee finishes execution and returns. The syntax for synchronous activations is similar to that used for Fortran 90 subroutine and function calls. Function methods may be only activated in a synchronous way.

The values returned by a call to a function method can be utilized in the usual way after the call is completed. Similarly, the values returned for dummy method arguments with intent **OUT** or **INOUT** are well-defined once the invoked method has completed execution.

Example 2.4.1: Synchronous Method Activation

The code fragment below illustrates how the methods defined for the buffer SDA of Example 2.3.2 may be synchronously activated in a program unit having access to two buffers.

```
SDA (buffer_type) :: a,b
REAL              :: x
. . .
CALL a.getb(x)     ! The first element of buffer 'a'
                   ! is retrieved and stored in 'x'

CALL b.putb(x)     ! Element 'x' is stored in buffer 'b'
                   ! The following statement is executed
                   ! after 'putb' has finished its execution.
. . .
```

In this code the first element of buffer `a` *is moved to buffer* `b`. *The* `getb` *method has to be activated synchronously in order to avoid the activation of* `putb` *with an undefined argument. However, the execution of the caller is also delayed by the synchronous call to* `putb`, *which might not be necessary in this example. We show below in Example 2.4.2 how the* `putb` *method can be activated asynchronously.*

■

2.4.2 Asynchronous Method Activation

An asynchronous method activation is generated as a result of executing a **SPAWN** statement. The caller continues its execution asynchronously with the callee and there is no implicit synchronization between them. Therefore any access to a variable passed to the invoked method with intent **OUT** or **INOUT** can result in a race condition. However, the language provides facilities for *explicit synchronization* as shown in Section 2.6.

Example 2.4.2: Asynchronous Method Activation

The modification of Example 2.4.1 illustrates how the `putb` *method can be activated asynchronously, thus enabling the parallel execution of the caller and the* `putb` *task. The keyword* `CALL` *preceding the* `getb` *activation may be omitted.*

```
SDA (buffer_type) :: a,b
REAL :: x
. . .
CALL  a.getb(x)
SPAWN b.putb(x)   ! putb is spawned as asynchronous task
                  ! executing in parallel with the caller.
. . .
```

■

Spawning of Procedures. The full specification of an SDA may imply a significant amount of redundant notation if an SDA does not take advantage of the data sharing facilities. In order to eliminate some of this redundant overhead, Opus allows procedures not declared within an SDA type to be directly spawned in the same manner as an asynchronous method activation. This means that there is no requirement in this case to explicitly define an associated SDA type and variable, and to explicitly create an SDA. All these actions are taken care of by the Opus compiler. Tasks generated by spawning a procedure are referred to as *procedure tasks*. An explicit interface is required for external procedures spawned as procedure tasks.

Example 2.4.3: Spawning of Procedures

Consider the specification of a producer generating data to be stored in our fifo buffer. The following code fragment outlines the required framework using the full SDA notation: first, an SDA type is defined containing only the `produce` *method. An SDA variable of this type needs to be declared and an SDA object is created by executing the intrinsic method* `CREATE` *associated with an* `ON` *clause describing the resources allotted to the SDA (cf. Section 2.4.3). After these steps, the* `produce` *method may be activated.*

```
SDA TYPE producer_type
CONTAINS
   SUBROUTINE produce(b,...)
     SDA (buffer_type) b
        . . .
   END SUBROUTINE produce
END SDA TYPE producer_type

. . .
SDA (producer_type) :: producer_variable
   . . .
CALL producer_variable.CREATE() ON (...)
   . . .
SPAWN producer_variable.produce(...)
```

Since this code does not exploit the data sharing facilities of SDAs it contains a significant amount of redundant notation. Using the possibility to directly spawn procedures, most of this redundancy can be removed. The resources allocated to the SDA can be directly specified when spawning the procedure using the same `ON` *clause. The code fragment below is semantically equivalent to the above program:*

```
SUBROUTINE produce(b,...)
   SDA (buffer_type) b
      . . .
END SUBROUTINE produce

. . .

SPAWN produce(...) ON (...)
```

■

2.4.3 Intrinsic Methods

Opus provides a set of intrinsic methods which are common to each SDA. These methods support the creation, saving, and termination of an SDA. Furthermore methods for accessing an SDA's internal data are provided.

2.4.3.1 Creating an SDA Object

Two methods for creating an SDA object are supported: The method **CREATE** is used to explicitly specify an object based on an SDA definition while **LOAD** constructs an SDA object by copying information from an external object which has been saved earlier.

An intrinsic function **CREATED** is provided that returns the creation state of an SDA variable, much like the Fortran intrinsic **ASSOCIATED** does for pointer variables. Recall that, unlike F90 pointers, SDA variables are default initialized such that **CREATED** returns **false** if applied to a variable that has not been modified after its declaration.

Example 2.4.4: SDA Creation

The following code fragment illustrates the creation of SDAs using our standard buffer example.

```
SDA (buffer_type) :: S1, S2(N,M), S3

IF (.NOT. CREATED(S1)) THEN
   CALL S1.CREATE(128)
END IF

DO I=1,N
   DO J=1,M
      CALL S2(I,J).CREATE(32*MAX(I,J))
   END DO
END DO

CALL S3.LOAD("persistent_buffer")
```

First, three SDA variables (`S1`*,* `S2`*, and* `S3`*) associated with SDA type* `buffer_type` *are declared.* `S2` *is a two-dimensional array of SDAs. All three variables are default initialized. Then, the creation status of* `S1` *is inquired by means of the* `CREATED` *intrinsic. If* `S1` *has not been associated with an SDA object, a new SDA object of type* `buffer_type` *with size 128 is created and the SDA variable* `S1` *refers to the new object. Then,* `N*M` *objects of type* `buffer_type` *are created in the following loop with their size depending on the loop variables* `I` *and* `J`*. Finally, a persistent SDA stored with the name* `persistent_buffer` *is restored and associated with* `S3` *by means of the* `LOAD` *intrinsic.*

■

The creation process takes typically five steps which might slightly differ for **CREATE** and **LOAD**. This will be discussed separately later.

Let v denote the SDA variable referred to by **CREATE** or **LOAD**. Then, based on the declaration of v and the arguments and attributes of the call, an object, B, is constructed by executing the following steps:

1. Determine an SDA type, t.
2. Perform resource allocation, as specified in the (optional) resource request.
3. Allocate data structures for B, as determined by t, the resource allocation, and distribution and alignment directives associated with t.
4. Initialize B.
5. Bind B to v.

The completion of the creation starts the *lifetime* of B. B is now accessible, via v, for activations of arbitrary methods associated with t, except for object creation. The lifetime of B terminates when **TERMINATE** or **KILL** (see Section 2.4.3.3) is activated.

If a creation fails for any reason, and a status clause has been specified, then the status variable is set to a non-zero value. Applying the **CREATE** (or **LOAD**) method to a variable which already has been bound to an SDA object results in a runtime error.

CREATE. The SDA variable, v, to which **CREATE** is applied, has to be typed and must not be bound to an object. Let tt denote the associated parameterized SDA type. Steps 1 and 4 from above are performed as follows:

- Step 1:
  For all dummy arguments of tt, actual arguments must be provided in the create statement. These arguments are bound to the corresponding dummy arguments. As a result of this, SDA type t is obtained.

- Step 4:
  Default initialization of B is performed, as required for a well-defined initial state. After this, the (optional) user-specified initialization method (cf. 2.4.3.2) is called.

LOAD. The SDA variable v may either be typed or generic, with type attribute tt. It may not be bound to an object. Steps 1 and 4 from above are performed as follows:

- Step 1:
  The load statement specifies (by means of a variable or a constant) a character string which identifies an external SDA object, x. Let t_x denote the SDA type of x. t_x must conform to tt (trivial if tt is of type *generic*). t is defined to be t_x.

- Step 4:
  The data structures of B are initialized by copying out the values of x.

Resource Request. Each SDA operates on a set of resources which are allocated at the time the SDA object is created, and deallocated at the time of its termination.

Different SDA objects may execute on disjoint or overlapping sets of resources. Note that the resource request as discussed below applies also to procedure tasks.

A resource request may consist of the *machine specification* and the *processor specification*. The *machine specification* identifies the (physical or logical) machine on which the SDA object is to be created. The machine can be specified either *directly* or *indirectly*. A *direct specification* identifies a machine by a string with a system-dependent meaning. An *indirect specification* provides the identification of an SDA (which must have been initialized) or of a procedure task (which must be active). Indirect machine specification associates the machine allocated to the given SDA with the newly created SDA. A procedure task is identified by its event variable. Note that an indirect machine specification using event variables is only possible when referring to procedure tasks, since they are not associated with an SDA. For the ease of programming the intrinsic attribute **SELF** identifies the resources allocated to the task creating the new SDA. **SELF** is also the default value used if the machine specification is omitted.

The *processor specification* can only be given directly. If it is omitted either all processors available on the specified machine (in case of direct machine specification) or the processors assigned to the given SDA (in case of indirect or default **SELF** machine specification) are devoted to the newly created SDA object. Direct processor specification uses the syntax and semantics of the HPF-processor declaration. It might be preceded by the keyword **NEW** or **SAME** if the machine specification was done indirectly. If **NEW** is used, then the specified processors are allocated in addition to those already associated with the given SDA; **SAME** is used to ensure that exactly the processors associated with the given SDA are allocated. If no allocation strategy is specified, the allocation of the processors is system dependent and they might be overlapping with those associated with the other SDA. An HPF processor name or processor section which is accessible by the caller may also be used to specify an SDA's processors. Obviously, the machine specification must be **SELF** in this case.

Example 2.4.5: Resource Request

Below we illustrate possibilities for a resource request using the standard buffer SDA.

```
      SDA (buffer_type) :: a,b
      . . .
      CALL a.CREATE(128)
! The SDA is created on the same resources allocated to the caller.

      CALL a.CREATE(128) ON (MACHINE='CS2')
! The SDA is created on the machine 'CS2' using all available processors
```

```
      CALL b.CREATE(128) ON (MACHINE = a, NEW PROCESSORS = 32)
! The SDA is created on the same machine as SDA 'a', using 32 processors
! which are disjoint from the processors associated with 'a'.

      CALL b.CREATE(128) ON (PROCESSORS = 32)
! The SDA is created on the same machine associated with the caller
! using 32 processors which might overlap with the processors
! allocated to the caller.

!HPF$ processors p(number_of_processors())
      CALL b.CREATE(128) ON (PROCESSORS = p(1:4))
! The SDA is created on the same machine associated with the caller
! using the first 4 processors of the virtual HPF processor array 'p'.
```

■

2.4.3.2 Initialization of an SDA Object

After the creation of an SDA object its internal variables are default initialized as required for a well-defined initial state. In addition to the default initialization the user can specify a parameterless subroutine method to be an *initialization method* (similar to C++ *constructors* [171]). The name of the initialization method must be the same as the SDA type name and the method must not yield any return value. The user-supplied initialization method is implicitly executed immediately after the object creation.

Syntactically, initialization methods are treated as normal subroutine methods, thus no additional syntax is required.

2.4.3.3 Termination of an SDA Object

The intrinsic method **TERMINATE** is used to terminate the lifetime of an SDA object. When calling **TERMINATE** the SDA variable must be bound to an object. The effect of this method activation is the deletion of the object.

The activation of **TERMINATE** is secure in the sense that it is ensured that all asynchronously spawned tasks must have finished their execution and all deferred method invocation requests must have been served. Only if no outstanding tasks exist for an SDA, **TERMINATE** will delete the object.

A more brute force alternative to **TERMINATE** is the intrinsic method **KILL**. An activation of **KILL** causes the immediate termination of the lifetime of an SDA object and its deletion. The execution of all outstanding tasks is stopped, all deferred requests are cancelled, and the object is deleted.

A successful completion of **TERMINATE** or **KILL** causes the SDA variable to which the command has been applied to become disassociated.

2.4.3.4 Saving an SDA Object

Opus provides facilities to make SDA objects persistent by saving them on external storage. An external name for the saved object might be supplied to the intrinsic method **SAVE**. The name might be omitted when the object was loaded from external storage. In this case the external name specified in the loading method is used. If an object with the specified name already exists, it is replaced with the object executing the method request. Otherwise a new object is created on the external storage.

When saving an object an exact image of the object's state in the time the method request is executed will be generated.

2.4.3.5 Access to an SDA's Data

Two intrinsic methods are provided for manipulating the data on an SDA. **PUT** is used to *generate* an SDA's internal variable, i.e. to assign it a value. Using the method **GET** an internal variable can be *consumed*, i.e. its value is retrieved. The syntax of **GET** and **PUT** is modeled on the Fortran 90 syntax for accessing derived types.

Each access using the standard Fortran 90 syntax for accessing a module's data is implicitly transformed to a **GET** or **PUT** method, respectively.

2.5 SDA Assignment and Equivalence

An SDA variable may become associated with an SDA object by one of three means:

1. activation of the **CREATE** or **LOAD** method;
2. if it is a dummy argument by passing an existing SDA as actual argument; or
3. through the execution of an *SDA assignment statement.*

Example 2.5.1: SDA Variable Association

The following code excerpt illustrates the three means for associating an SDA variable with an SDA object.

```
...
SDA (buffer_type) :: a

CALL a.CREATE(...)
CALL sub(a)
...

SUBROUTINE sub(x)
   SDA (buffer_type) :: x
   SDA (buffer_type) :: y

   ...
   y = x
   ...
END SUBROUTINE sub
```

First, an SDA variable `a` *is declared and SDA object* `o`*, which is bound to variable* `a`*, is created by applying the* `CREATE` *method to* `a`*. Subsequently,* `a` *is passed as input argument to subroutine* `sub` *resulting in an association of the dummy variable* `x` *with* `o`*. Finally, the execution of the SDA assignment statement* `y = x` *results in an association of* `y` *with* `o` *so that after the execution of this statement all three SDA variables (*`a`*,* `x`*, and* `y`*) refer to the same SDA object* `o`*.*

■

SDA Assignment Statement. Recall that an SDA variable is conceptually a pointer to an SDA object; therefore, SDA assignment statements are semantically equivalent with Fortran pointer assignment statements. Specifically, an assignment statement whose left-hand-side (lhs) is an SDA variable results in an association of the lhs variable with the SDA object associated with the right-hand-side (rhs), unless the rhs is not yet associated with an SDA object. If the rhs is not an SDA variable but an expression, the expression must yield an SDA variable. Unless the lhs of an SDA assignment statement is a generic SDA variable, the type of the rhs object must conform with the type of the lhs variable.

SDA Equivalence. As we have seen from the example above, multiple SDA variables may refer to the same SDA object. Opus provides a mechanism for querying whether two SDA variables are associated with the same object or not—the intrinsic SDA function **EQUAL**.

EQUAL receives a scalar SDA variable as input and returns **true** if the SDA object to which **EQUAL** is applied is the same one as the object associated with the input variable; otherwise **false** is returned.

Example 2.5.2: SDA Equivalence

The following code excerpt illustrates a simple SDA for various equivalence tests.

```
SDA test(b)
  SDA (buffer_type) :: b
CONTAINS
  SUBROUTINE test_equal(x,y)
     SDA (test) :: x
     SDA (buffer_type) :: y

     IF (EQUAL(x)) ...
     IF (b.EQUAL(y)) ...
  END SUBROUTINE test_equal
END SDA test
```

The first equivalence test `EQUAL(x)` *returns* **true** *if dummy variable* `x` *is associated with the object within which the call is executed. The second test* `b.EQUAL(y)` *queries whether SDA variable* `b` *and* `y` *are associated with the same SDA object.*

■

2.6 Synchronization

2.6.1 SDA Synchronization

As discussed in Section 2.3, SDAs provide implicit and explicit synchronization facilities to ensure a coherent state of their data. Implicit synchronization is achieved by allowing only one method of an SDA to be active at a time. This method has exclusive access to the whole SDA and is an atomic operation. This feature is similar to monitors [109] in shared memory languages.

The possibility of assigning a condition clause (the **WHEN** construct) to an SDA's method allows explicit synchronization. A method may only be activated if its associated condition clause evaluates to **true**.

2.6.2 Task Synchronization

Asynchronously executing tasks do not provide any implicit synchronization. This means that for instance the caller could access variables passed with intent **OUT** or **INOUT** to an asynchronously spawned task before the task finishes its execution. This access can result in a race condition.

The language provides *event variables* to explicitly synchronize asynchronous tasks. A new type **EVENT** is introduced and all variables of that type must be explicitly declared. When a declaration of an event variable, say E, is processed, then E is implicitly set to *undefined*. No explicit initialization is possible in the declaration.

An asynchronous task is identified by an *event value* which has a *completion state*. This completion state determines whether or not the task has completed execution and is set to **false** at the creation. If an event clause with event variable E is specified as part of the spawn statement, E is bound to the task's completion state. E must be *undefined* when spawning the task.

The event variable E remains bound to the task's completion state, until all synchronization statements referring to E have completed their execution. Then E is set to *undefined* and can be used again. Similarly, a task's *event value* has to be active while there exists any event variable bound to it. Any access to undefined events results in a runtime error, unless a status clause has been specified.

An event variable can be accessed using the **WAIT** statement or the private intrinsic functions **WAITANY** and **TEST**:

Wait. The execution of a wait statement, **WAIT**(E) blocks the executing task until the method associated with E has finished its execution. If E is not bound to an event value, thus its value is *undefined*, **WAIT** terminates unsuccessfully. If the method execution is already completed at the time **WAIT** is executed, then its effect is empty.

At the time a wait statement is successfully completed, it is guaranteed that the values of all dummy arguments with intent **OUT** or **INOUT** of the associated method execution are updated.

Similar to **WAIT** the execution of the private intrinsic function **WAITANY**(*event-variable-list*) blocks the executing task until *any* method associated with $E \in$ *event-variable-list* has finished its execution. The position of E in *event-variable-list* is returned.

Test. A nonblocking alternative to the wait statement is the private intrinsic function **TEST**(E). The execution of the test function returns the actual value of the event value, to which E is bound, thus allowing the caller to continue execution. If the value of E is *undefined*, the test function terminates unsuccessfully.

Example 2.6.1: Task Synchronization

The synchronization needed for the copy between buffers illustrated in Example 2.4.2 can also be achieved by using `WAIT` *as shown below. A similar effect can be gained by actively polling (testing) an event within a while-condition.*

```
. . .
SDA (buffer_type) :: a,b
REAL :: x
EVENT :: e1,e2
. . .
e1=SPAWN a.getb(x)
WAIT(e1)                   ! Block execution until the getb-method
                           ! has finished its execution
e2=SPAWN b.putb(x)
DO WHILE (.NOT. TEST(e2))  ! Execute the while body as long as
   . . .                   ! the putb-method has not finished
                           ! its execution
END DO
```

■

2.7 Discussion

After having introduced the salient features of the Coordination Language Opus in the preceding sections of this chapter we now take a closer look at Opus' support for DHPC applications. Specifically, we review the features of Opus by means of the criteria presented in Chapter 1.

Heterogeneity. While the base language of Opus, HPF, offers only limited language support for dealing with heterogeneous systems (see also Chapter 7), Opus has been specifically designed for heterogeneous architectures. In particular, Opus allows dynamic creation of objects (SDAs) on user specified resources—the user may specify the (logical or physical) machine as well as the number of processors on which an SDA should be created. This functionality gives the user the means required for performing an appropriate mapping among the physical resources in a complex cluster and computational modules that are encapsulated within an SDA. However, the mapping of SDAs is, although performed at runtime, invariant; i.e. once created SDAs cannot be re-located which is a major problem for applications that require dynamic load-balancing. This restriction will be lifted in Chapter 3.

In addition to the language features that allow a design of heterogeneity aware programs, advanced runtime support is required for an efficient interaction among SDAs located on heterogeneous machines. This issue, which has also been discussed in [133], will be elaborated in Chapter 5.

Interoperability. Vertical interoperability is ensured in Opus through HPF's **EXTRINSIC** mechanism that allows to escape from HPF (and thereby from Opus) by calling routines written in different languages, such as C, or employing different paradigms, such as explicit message passing. In addition to this elaborated vertical interoperability mechanism of HPF, Opus enhances HPF's horizontal interoperability capabilities. The HPF standard does not define appropriate interfaces for an invocation of HPF routines from systems others than HPF. Opus, instead, wraps HPF routines within SDA methods and allows them to be invoked from any other language for which an Opus binding exists. In other words, the SDA mechanism of Opus builds the common abstraction that facilitates interactions among different languages and paradigms. Specifically, Opus allows HPF routines to be invoked from simple Fortran programs (through its HPF/Fortran binding) and also from Java programs (through its Java binding which is presented in Chapter 4).

Parallelism and Communication. Opus combines fine grained data parallelism, expressed in HPF, with coarse grained task parallelism in a unified, hybrid parallel model. Moreover, nested task parallelism is expressible by means of HPF's task parallel features. Communication between data parallel tasks is accomplished by means of invoking methods in shared objects. Opus provides both synchronous and asynchronous method invocation and efficient means for synchronizing parallel tasks. In particular, SDAs are self-synchronizing in that they have monitor semantics and that executions of their methods may be guarded by a condition clause. Synchronization with asynchronous tasks is accomplished with help of events associated to them.

However, communication between Opus tasks requires complicated and expensive method invocation within shared SDAs and so the size of the cooperating tasks must be large enough to amortize this cost. Therefore, Opus is not well-suited as a programming language for fine-grained task parallel applications. Strategies for reducing this overhead will be discussed in Chapter 3.

Summary. In contrast to many other systems (cf. Chapter 7), Opus provides a unified, high-level, hybrid parallel model that integrates fine grained data parallelism with coarse grained task parallelism. Apart from this high-level hybrid parallel model which is the major contribution of Opus, basic support for heterogeneous systems as well as horizontal and vertical interoperability is provided. Therefore, Opus can be regarded as a practicable tool for implementing DHPC applications.

Nevertheless, Opus also has some shortcomings, in particular with respect to dynamic task migration and fine grained task parallel problems. In the next chapter we develop some extensions to the basic language that are aimed at removing most of the deficiencies discussed above.

Chapter 3

Advanced Opus Features

3.1 Introduction

In the previous chapter we have presented the salient features of the Coordination Language Opus, in particular its hybrid parallel model combining task and data parallelism. We also identified some shortcomings of Opus, especially its expensive method invocation overhead and its lack of support for dynamic task migration.

In this chapter we develop new features that are designed to remove most of these problems. Specifically, we provide means that reduce the method invocation overhead on the one hand, and allow an exploitation of more flexible and fine grained task parallelism on the other hand. Furthermore, features for migrating SDAs are developed. We briefly summarize these features below.

Intra-SDA Parallelism. We present a new compiler directive called *ParBlock* that allows the specification of intra-SDA parallelism on the method level. Concurrent executions of methods within an SDA provide a more flexible means for task parallelism in that idle times due to blocked tasks may be overlapped with useful computation. ParBlocks allow a simple and intuitive specification of concurrency among methods while preserving Opus' monitor semantics as the default mechanism.

Nested Composition of SDAs. A recent trend in software engineering is to build reusable software components [178] by means of object based component frameworks that "wrap" reusable computational modules and provide a clean interface to them. The efficient exploitation of hybrid forms of parallelism within and across components is one of the key issues in the design of high performance component frameworks. While Opus provides the required hybrid parallel model as well as a clean interface, it is not well suited for fine grained or dynamically nested task parallel applications. In particular, the creation of an SDA is a heavyweight process that involves the establishment of a new SDA context; invoking a method of an SDA is also a quite heavyweight and expensive process that involves, among others, data copying. Thus, standard component composition technologies such as containment or aggregation are often too expensive in Opus. In order to cope with these problems we introduce *lightweight* SDAs that may exist only within the context of a standard (heavyweight) SDA with which they share resources.

SDA Migration. An important aspect of DHPC applications is their underlying hardware architecture—complex clusters. As discussed in Chapter 1 any system dealing with complex clusters has to provide means for tackling the heterogeneity of hardware, in particular means for dynamically mapping computation to system components are required.

Opus provides some limited support with this respect in that SDAs are dynamically created on user specified resources; however, once created SDAs cannot be re-located. In this chapter we propose a new intrinsic method *migrate* that allows a dynamic re-location of SDAs during program execution and thus facilitates, among others, dynamic load-balancing strategies.

Priority Scheduling. Opus provides *condition clauses* with which the user can influence the scheduling of method executions dependent on the status of the internal data of the SDA. For fine-tuning of tasks it is however necessary to schedule method executions independently of any other conditions. For instance, the user might wish the *migrate* method introduced above to be executed immediately after its invocation, independent of other methods that are ready for execution. We present a *priority clause* which provides the necessary functionality.

In the remainder of this chapter we give a detailed description of all of these features: Section 3.2 introduces ParBlocks; lightweight SDAs are presented in Section 3.3; in Section 3.4 we deal with dynamic migration of SDAs and in Section 3.5 we discuss priority scheduling within SDAs. We end the chapter with a summarizing discussion in Section 3.6.

3.2 Intra-SDA Parallelism

3.2.1 Introduction

Opus SDAs have monitor semantics, thus only one method may be active in an SDA at a given time and this method has exclusive access to the data of the SDA. With monitor semantics a consistent state of the SDA data is ensured at the expense of potential parallelism losses. In particular, all method invocation requests arriving at an SDA are served in a sequential manner. As a result of this sequential processing of requests, an SDA that is shared among many SDAs which frequently emit method invocation requests may easily become a bottleneck.

The possibility of serving multiple requests concurrently helps in overcoming this problem. Thereby, an additional potential for parallelism within the standard Opus mechanisms is introduced. This parallelism can for instance be exploited on shared memory systems—but also on systems with distributed memory idle times that are due to communication or synchronization with other tasks can be overlapped with useful computation, thus reaching a better utilization of the available computation nodes. Moreover, all methods of an SDA share the same address space and have thus access to the same data. Therefore, data exchange between methods of an SDA is much less expensive than between methods of different SDAs. This fact is of particular importance in pipelined computations, as we will show below.

At runtime, intra-SDA parallelism is enabled through the provision of multiple SDA threads (cf. Chapter 2) within the context of an SDA. All of these SDA threads run concurrently and may execute methods in response to method invocation requests.

However, allowing concurrent executions of multiple methods within an SDA poses a number of difficulties (see for instance [143] for a detailed discussion of intra-object concurrency) among which the most important one is how to specify potential parallelism and needed synchronization among methods. A common, low level approach which is employed in multithreading systems is to protect accesses to shared data with appropriate *mutex* variables. This approach is taken in pthreads [114], where the user has to deal with mutex variables explicitly, as well as in Java [7], where an implicit mutex is provided for any object. Jade [164] takes a different approach in not protecting the data directly, but forcing the user to specify all accesses to shared data; the compiler then inserts the required protection based on these access specifications.

In Opus we take a higher level approach and relieve the user from dealing with single data protection. We believe that the proper granularity of parallelism within SDAs is the method level. A more fine grained level of synchronization, such as direct data protection via mutexes, would require advanced analysis in order not to interfere with the additional synchronization imposed by the underlying HPF system.

Given the concurrency granularity being the method level, we may generally state that methods can safely execute concurrently if they do not interfere [5]. Compiler analysis can be used for detecting some potential for intra-SDA parallelism, however, a compiler is generally not able to detect all cases and therefore some support from the user is needed in order to exploit intra-SDA parallelism to a greater extent.

It is worth noting that our concept of intra-SDA parallelism refers solely to concurrent method executions as a result of method invocation requests. Concurrency within a method execution may be exploited by means of the data and task parallel features of HPF.

In this section we propose a compiler directive called *ParBlock* which can be used for specifying potential parallelism among methods in a simple and intuitive way.

3.2.2 ParBlocks

The ParBlock directive is a compiler directive composed of the Opus compiler directive origin (**!OC$**) and the **PARBLOCK** keyword followed by a semi-colon separated list of parenthesized method-name sets, called *par-sections*. All methods, whose names are part of the same par-section are guaranteed to be interference free and may thus be executed concurrently. Multiple ParBlocks may be specified for an SDA type and a method name may occur multiple times within a ParBlock as well as in multiple ParBlocks. If a method name occurs at least two times within the same par-section, multiple instances of this method may execute concurrently; if a name is present more than two times within a par-section the notation is redundant and has no additional semantics. Any two methods that do not jointly belong to a par-section are not guaranteed to be interference free and, therefore, have to be executed in a mutually exclusive manner. Hence, a method that is not referenced in any ParBlock has still exclusive access to an SDA.

In the following, we introduce the syntax of ParBlocks using two synthetic examples.

Example 3.2.1: ParBlocks

Consider an SDA with 6 methods, `a, b, c, d, e,` *and* `f`. *Assume that all methods are allowed to execute in parallel but with the restrictions that (1) method* `b` *and* `c` *cannot execute concurrently and (2) method* `d` *cannot execute concurrently with either* `e` *or* `f`.

There are several possibilities of how to express this relationship by means of a ParBlock. One way would be to specify all pairwise interference free methods explicitly:

```
!OC$ PARBLOCK (a,b);(a,c);(a,d);(a,e);(a,f);(b,d);(b,e);(b,f); &
              & (c,d);(c,e);(c,f);(e,f)
```

It is, however, possible to name more than two methods within a par-section, hence the following ParBlock is semantically equivalent with the one given above:

```
!OC$ PARBLOCK (a,b,e,f);(b,d);(a,c,d);(c,e,f)
```

Consider another SDA with 4 methods, `a, b, c,` *and* `d`. *We want to specify partial parallelism such that* `a` *is allowed to execute in parallel with* `b` *and* `c`, *and* `c` *is allowed to execute in parallel with* `d`. *All other combinations cannot be executed in parallel.*

The ParBlock directive expressing this partial parallelism is

```
!OC$ PARBLOCK (a,b);(a,c);(c,d)
```

■

In an early proposal [126] ParBlocks were not only composed of par-sections, but also of so-called *sync-sections* which were used to express the need for synchronization among methods. That proposal was inspired by *Path Expressions* [37, 38] which provide an elegant means of specifying synchronization between processes by describing how a process is allowed to execute in relation to others, irrespective of their invocation order.

However, since mutual exclusive execution is the default semantics of method executions within an SDA and condition clauses provide a graceful mechanism for synchronization, all that is really needed is some means for specifying potential parallelism—par-sections.

Opus, being an extension to HPF, may also exploit parallelism information provided by HPF and Fortran. Specifically, the Fortran 95 **PURE** attribute asserts that a procedure is free of certain side effects that prevent concurrent execution. Hence, all pure procedures of an SDA type are treated as if they were named within a par-section. Moreover, multiple instances of a pure procedure may safely execute concurrently, as well.

Example 3.2.2: ParBlocks and Pure Procedures

Consider the following SDA type:

```
SDA TYPE foo
   ! data
CONTAINS
   PURE FUNCTION f(...)
      . . .
   END FUNCTION f

   PURE SUBROUTINE sub(...)
      . . .
   END SUBROUTINE sub
END SDA TYPE foo
```

Although no explicit ParBlock is specified within this SDA type, both procedures are `PURE` *and may be executed concurrently—as if the following ParBlock would be present:*

```
!OC$ PARBLOCK (f,f,sub,sub)
```

■

ParBlocks are parsed by the Opus compiler (cf. Chapter 5) which generates a so-called *ParSet* for every method declared within an SDA type. A ParSet contains the names of all the methods with which the associated method may safely execute concurrently. By default, the ParSet of a method is empty and thus the method has exclusive access to the SDA. Figure 3.1 shows the ParSets for the first SDA type of Example 3.2.1 and for the SDA type of Example 3.2.2. The algorithm for generating these sets is presented in Chapter 5.

ParSets and condition clauses are used together in order to evaluate whether a method invocation request is ready for execution or not. In particular, before launching the execution of a method it is necessary to check if

1. the method is allowed to execute in parallel with all other methods currently being executed within the SDA, and if
2. its condition clause is satisfied.

In other words, a method invocation request is said to be *ready for execution* if and only if the associated condition clause is satisfied and the set of all active methods of the SDA is a subset of its ParSet. Obviously, both checks have to form an atomic action and checks for different requests have to be executed mutually exclusively in order to avoid inconsistencies due to overlapping checks.

Summarizing the above, Opus provides a set of features for specifying intra-SDA parallelism and synchronization, both dependent and independent of the SDA's internal state:

- **condition clauses:** for specifying dynamic synchronization based upon the internal state of the SDA,

Method	ParSet
a	b, c, d, e, f
b	a, d, e, f
c	a, d, e, f
d	a, b, c
e	a, b, c, f
f	a, b, c, e

(a)

Method	ParSet
f	f, sub
sub	sub, f

(b)

Figure 3.1: ParSets for Example 3.2.1 (a) and Example 3.2.2 (b)

- **ParBlocks:** for specifying parallelism independently of the SDA's internal state, and
- **pure attributes:** for specifying potential parallelism according to the Fortran 95 standard.

There are some subtleties that have to be considered when using ParBlocks, in particular with respect to fairness of method execution and potential life-locks. Consider three methods `a`, `b`, and `c` with the following ParBlock: `(b,c)`. If methods `b` or `c` are invoked frequently, there might be little chances for `a` to ever start executing since it must wait until no other method is being executed. This situation can be overcome by specifying proper condition clauses for `b` and `c` which will prevent them from executing at times, thus giving `a` a chance for being executed.

In the following case study we illustrate how parallelism information specified in ParBlocks and synchronization by means of condition clauses work together in order to obtain intra-SDA pipelines. A comparison of ParBlocks with Java's synchronization features can be found in [127].

3.2.3 Case Study: FFT

An important class of applications that can take advantage of intra-SDA parallelism are pipelined problems to be found for instance in FFT computations. A FFT code is typically structured into four tasks (*input, column-fft, row-fft, output*). See Chapter 6 for a more detailed discussion of FFT. Each of those components could be scheduled on a separate stage of a pipeline. Although such pipelines can be expressed in Opus by assigning each stage to a separate SDA, this kind of structure imposes significant data transfer overhead among the SDAs. The overhead can be reduced by overlapping I/O with computation within an SDA in a pipelined manner.

The code excerpt given in Example 3.2.3 sketches the declaration of two SDAs implementing a two stage pipelined FFT. The first SDA (`fft_stage1`) is responsible for *input* and *column-fft*, while the second SDA (`fft_stage2`) performs the remaining tasks *row-fft* and *output*.

Within each SDA, an intra-SDA pipeline is exploited to overlap I/O operations with useful computation. Apart from the necessary parallelism information given via ParBlocks, a separate copy of the FFT data has to be allocated for each stage of the pipeline in order to guarantee interference freedom. Finally, the correct pipelined behavior is guaranteed by appropriate condition clauses.

In the main program (or driver routine) an object of each type is created on disjoint resources and a reference to the second stage object is passed to the object representing the first stage. Subsequently, the methods of the first SDA are invoked asynchronously. Note that the SDA is self-synchronizing and no synchronization on the level of the driver routine is needed. Similarly, method `col_fft` asynchronously spawns the activities of the second stage which is also self synchronizing. Both routines offer a routine `wait_finished` which allows synchronization on program termination. Specifically, `wait_finished` in the second stage SDA is associated with a condition clause such that it may only be executed if all iterations have been performed. `wait_finished` in the first stage

SDA is associated with an according condition clause and invokes `wait_finished` in the second SDA. Hence, a call to `wait_finished` issued from within the driver routine may only return if all data has been processed in both stages of the pipeline.

Example 3.2.3: Pipelined FFT

```
PROGRAM FFT

!***** STAGE 1 *****
  SDA TYPE fft_stage1(s2,max_iteration)
    SDA (fft_stage2) :: s2
    INTEGER :: max_iteration

    ! local data
    TYPE (fft_data) :: data(0:1)
    ! flags used in condition clauses
    LOGICAL :: read_allowed = .TRUE.
    LOGICAL :: col_allowed = .FALSE.
    ! data set index
    INTEGER :: active_read = 0
    INTEGER :: active_col = 0
    ! iteration counter
    INTEGER :: passed_iteration = 0

  CONTAINS

    SUBROUTINE read_data() WHEN (read_allowed)
      read_allowed = .FALSE.
      ! read fft_data using data(active_read)
      active_read = MOD(active_read+1,2)
      col_allowed = .TRUE.
    END SUBROUTINE

    SUBROUTINE col_fft() WHEN (col_allowed)
      col_allowed = .FALSE.
      read_allowed = .TRUE.
      ! compute column_fft(data(active_col))
      SPAWN s2.row_fft(data(active_col))
      SPAWN s2.output()
      active_col = MOD(active_col+1,2)
      passed_iteration = passed_iteration+1
    END SUBROUTINE

    SUBROUTINE wait_finished() WHEN (passed_iteration .eq. max_iteration)
      CALL s2.wait_finished()
    END SUBROUTINE

  !OC$ PARBLOCK(read_data, col_fft)

  END SDA TYPE fft_stage1
```

```
!***** STAGE 2 *****
  SDA TYPE fft_stage2(max_iteration)
    INTEGER :: max_iteration

    ! local data
    TYPE (fft_data) :: data(0:1)
    ! flags used in condition clauses
    LOGICAL :: row_allowed = .TRUE.
    LOGICAL :: output_allowed = .FALSE.
    ! data set index
    INTEGER :: active_row = 0
    INTEGER :: active_out = 0
    ! iteration counter
    INTEGER :: passed_iteration = 0

  CONTAINS
    SUBROUTINE row_fft(input) WHEN (row_allowed)
      TYPE (fft_data), INTENT(in) :: input
      row_allowed = .FALSE.
      ! compute row_fft(data(active_row)=input)
      active_row = MOD(active_row+1,2)
      output_allowed = .TRUE.
    END SUBROUTINE

    SUBROUTINE output() WHEN (output_allowed)
      output_allowed = .FALSE.
      row_allowed = .TRUE.
      ! call output(data(active_out))
      active_out = MOD(active_out+1,2)
      passed_iteration = passed_iteration+1
    END SUBROUTINE

    SUBROUTINE wait_finished() WHEN (passed_iteration .eq. max_iteration)
    END SUBROUTINE

  !OC$ PARBLOCK(row_fft, output)
  END SDA TYPE fft_stage2

!***** DRIVER *****
  SDA (fft_stage1) :: s1
  SDA (fft_stage2) :: s2
  INTEGER max_data = ...

  CALL s2.create(max_data) ON (...)
  CALL s1.create(s2,max_data) ON (...)
  DO i = 1,max_data
     SPAWN s1.read_data()
     SPAWN s1.col_fft()
  END DO
  CALL s1.wait_finished()

END PROGRAM
```

■

3.3 Nested Compositions of SDAs

3.3.1 Introduction

As we have described in Chapter 2 Opus is well suited for coordinating coarse grained data parallel modules, for instance components of environmental models or multi-disciplinary applications, by providing a well defined interface through shared data abstractions. These abstractions, however, imply expensive overheads that make Opus not well suited for fine grained or dynamically nested task parallel applications.

However, many scientific and engineering applications need to exploit a fine-grained level of task parallelism that requires an efficient means of task interaction since their underlying models are based on small, yet dynamically changing components organized for instance in graphs or trees. Examples of such algorithms include parallel search algorithms such as depth-first search and branch-and-bound [123] or oct-tree based n-body algorithms such as the Barnes-Hut method [14]. While such tree structures can be easily modeled with help of SDAs, the implied overhead will probably be prohibitive.

Specifically, SDA creation is an expensive task that involves, among others, the establishment of a new SDA context together with its (distributed) SDA threads. SDA interaction often requires data copying due to the copy-in/copy-out semantics of SDAs even if two communicating SDAs reside on the same physical processor(s). Finally, SDAs are always active objects that are associated with a separate thread of control although in most object-oriented or object-based systems there are many fewer asynchronously executing concurrent activities than there are objects. This is a particular problem in case of synchronous method invocation which requires the interaction of two threads of control instead of a simple procedure call that is handled by only one thread of control.

Lightweight SDAs are designed to remove most of the problems mentioned above. In particular, they are *passive* objects, i.e. objects that are not associated with a separate SDA thread, whose methods may be invoked with standard Fortran/HPF argument passing semantics. Nevertheless, lightweight SDAs are still objects that may be remotely accessed. In order to facilitate remote accesses in absence of a dedicated SDA thread, lightweight SDAs are associated with a standard SDA and use the runtime environment (the *context*) of this standard (*heavyweight*) SDA which is called the *base SDA* of the lightweight SDA.

3.3.2 Lightweight SDAs

In this section we develop the syntax and semantics of lightweight SDAs. Since lightweight SDAs are a modification of the standard SDA mechanism introduced in Chapter 2 we focus on the differences between lightweight SDAs and standard SDAs. Specifically, if we do not specify certain semantic aspects the corresponding semantics of standard SDAs apply to lightweight SDAs. In the following, we use the generic term *SDA* to refer to both, standard and lightweight SDAs.

3.3.2.1 Type Definition

Lightweight SDAs are defined by annotating the standard SDA type definition with the attribute **LIGHTWEIGHT**. Lightweight SDA types consist of:

- a name and a list of dummy arguments which can be used to parameterize the SDA type,
- a specification part, declaring the required types and data structures together with their distributions and other attributes, and
- a method part which specifies methods acting on the SDA's internal data structures.

Specification Part. The specification part declares the internal data structure of a lightweight SDA in the same way as the specification part of standard SDAs. In particular, depending on the accessibility attribute (**PUBLIC**, **PRIVATE**, or **READ_ONLY**), a variable declared within the specification part of a lightweight SDA may be accessed by any SDA having access to the SDA and/or from within the methods of the SDA. HPF directives may be used for specifying the distribution of the internal data.

Method Part. Methods are arbitrary procedures (either functions or subroutines) and may have arguments of any intent which are passed with standard Fortran 90/HPF semantics. This is a major difference to standard SDAs where method arguments are passed with copy-in/copy-out semantics.

Dummy arguments may be distributed using the rules applicable to procedure arguments in HPF. Furthermore data parallel statements and directives of HPF may be used to express data parallelism.

Note that within the body of a procedure only variables declared in the specification part of the SDA as well as all dummy arguments are accessible. Variables declared in other SDAs, in particular in the base SDA hosting the lightweight SDA, are only accessible via explicit or implicit method calls if a reference to the SDA is available.

Rationale. A major difference between lightweight SDAs and standard SDAs is the argument passing semantics for its methods. Standard Fortran/HPF semantics often allow call-by-reference instead of copy-in/copy-out and thus reduce the copy overhead.

Methods of lightweight SDAs do not have direct access to any data not declared within their specification and methods part or passed as input arguments. This is to ensure a consistent state of the data. However, accesses to data passed to asynchronously spawned methods may result in race conditions within both, the caller and the callee. Therefore, proper synchronization is necessary as will be discussed below.

3.3.2.2 Lightweight SDA Declaration

Lightweight SDA variables may be declared and used in the same way as standard SDA variables.

In particular, lightweight SDA variables may be passed as procedure arguments, thus making them accessible from outside of their base SDAs. Recursive declarations of lightweight SDAs are allowed which facilitates dynamically nested structures such as trees or lists.

Example 3.3.1: Lightweight SDA Declaration

```
SDA TYPE, LIGHTWEIGHT :: node
   ! specification part
   SDA (node) :: next
CONTAINS
   ! method part
END SDA TYPE node

SDA (node) :: list
```

SDA type `node` *contains a recursive reference (*`next`*) that allows a dynamic growth of the SDA list. Subsequently, variables of type* `node` *can be declared, such as variable* `list`*, which becomes the head of a list.*

∎

3.3.2.3 Lightweight SDA Creation

Lightweight SDA objects are created, i.e. lightweight SDA variables are instantiated, through the execution of the intrinsic method **CREATE** or **LOAD**. In the creation process, there are some fundamental differences between standard and lightweight SDAs:

- *Base SDA*: The creation of a lightweight SDA may only be performed from within another SDA whereas standard SDAs may be created also from within standard Fortran/HPF programs. The creation of a lightweight SDA establishes a parent/child relationship among the activator of the creation and the newly created lightweight SDA. This relationship is reflected in a tree structure called *SDA Tree* whose root is a standard SDA and all of its nodes are lightweight SDAs. A standard SDA building the root of an SDA tree is called the *base SDA* for all the lightweight SDAs of the tree.
- *SDA context*: Lightweight SDAs do not have their own SDA context but share it with their base SDA. In particular, their data structures are allocated within the context of the base SDA and all control activity related to the lightweight SDA is handled by the SDA threads of the base SDA.
- *Nodes*: The nodes on which a lightweight SDA is to be created must be a subset of the nodes allotted to its base SDA.

Rationale. In contrast to standard SDAs, which are *active* objects, lightweight SDAs are *passive* objects. Hence, lightweight SDAs do not have their own SDA context, but share it with their base SDA. In this respect, lightweight SDAs are comparable with objects in object oriented languages such as C++ or Java, which are created within the context of the calling process/thread. Lightweight SDAs are, however, not only pure passive objects, but *shared* objects that may be accessed remotely. While for instance in Java remote objects are facilitated by appropriate skeletons, in Opus this functionality is provided by the base SDA.

The restriction that lightweight SDAs may only be created within the context of a base SDA makes it obvious that the nodes onto which the lightweight SDA should be created have to be a subset of the nodes allotted to its base SDA.

As discussed in Chapter 2 the creation of a standard SDA object B bound to an SDA variable v takes five steps:

1. Determine an SDA type, t.
2. Perform resource allocation, as specified in the (optional) resource request.
3. Allocate data structures for B, as determined by t, the resource allocation, and distribution and alignment directives associated with t.
4. Initialize B.
5. Bind B to v.

Step 2 of above involves the creation of an SDA context, specifically of the SDA threads. By re-using the SDA context of the base SDA the creation process of a lightweight SDA consists only of steps 1, 3, 4, and 5 of above. The quite expensive step 2 is omitted.

It is worth noting that a lightweight SDA object becomes part of the SDA Tree of a heavyweight SDA only by being created from within an SDA already part of the tree. In particular, the membership to a specific SDA Tree is independent of the lexical scope of the lightweight SDA variable declaration. Thereby, lightweight SDAs differ from *inner classes* found for example in Java [7]. While inner classes are defined within the lexical scope of a base class, allowing full access to the data of the base class, lightweight SDAs have their own lexical scope. Hence, although created within the context of a base SDA, lightweight SDAs do not have direct access to the data of their base SDA. Note that the termination of a base SDA implies the termination of all lightweight SDAs being part of its SDA Tree since the termination of a standard SDA causes the deletion of its context.

Example 3.3.2: Lightweight SDA Creation

The following code segment illustrates the creation of lightweight SDAs based on the list type introduced in Example 3.3.1.

```
PROGRAM main
 SDA TYPE, LIGHTWEIGHT :: node
  ! specification part
  SDA (node) :: next
  ...
 CONTAINS
  ! method part
  SUBROUTINE new_node()
    IF (.NOT. CREATED(next)) THEN
      CALL next.CREATE()
    ELSE
      CALL next.new_node
    END IF
  END SUBROUTINE new_node()

  SUBROUTINE do_work(a,b)
    REAL, INTENT(IN) :: a
    REAL, INTENT(OUT) :: b
    REAL :: c

    c = ... a ...
    IF (CREATED(next)) &
        CALL next.do_work(c,b)
    ...
  END SUBROUTINE do_work
 END SDA TYPE node

 SDA TYPE :: list_base
 CONTAINS
  SUBROUTINE new_list(l)
   SDA (node), INTENT(INOUT) :: l
   CALL l.CREATE()
  END SUBROUTINE new_list
 END SDA TYPE list_base

 ! SDA declaration
 SDA (list_base) :: base
 SDA (node) :: list,list1, &
               wrong_list
 ! SDA creation
 CALL base.CREATE() &
               ON (MACHINE='foo')

 ! method invocations
 CALL base.new_list(list)
 CALL base.new_list(list1)
 CALL list.new_node()
 CALL list.do_work(a,b)

 ! illegal SDA creation
 CALL wrong_list.create() &
               ON (MACHINE='foo')
END PROGRAM main
```

Since lightweight SDAs must be ultimately based on heavyweight SDAs, a heavyweight SDA `list_base` *is introduced and created on a machine named "foo". SDA lists can now be created from within this* `list_base` *via the method* `new_list`*. Although variable* `list` *is declared within the* `main` *program, there is no parent/child relationship between* `main` *and* `list`*, but only between the heavyweight* `base` *and* `list`*.*

Multiple lightweight SDAs may be created within a heavyweight one (in this example `list` *and* `list1`*) resulting in sharing of resources among them.*

The activation of the `new_node` *method results in the creation of a new lightweight SDA nested in another lightweight SDA in a linked list like fashion. The last statement of the main program (the creation of* `wrong_list`*) is, however, illegal because the creation does not take place within an SDA.*

Methods of the lightweight SDA, such as `do_work`*, may be invoked from outside the SDA Tree (such as from within the main program) or from within the SDA Tree (such as from within method* `do_work`*). In the latter case a recursive descent along the SDA Tree can be modeled.*

Figure 3.2 depicts the SDA Tree that results from the execution of the above program.

■

3.3.2.4 Method Invocation

Methods of lightweight SDAs may be invoked in a synchronous or asynchronous manner, just as methods of standard SDAs. However, there is a major difference in the argument passing semantics. While arguments to methods of standard SDAs are passed with copy-in/copy-out semantics, arguments to methods of lightweight SDAs conforming to Fortran 90/HPF conventions

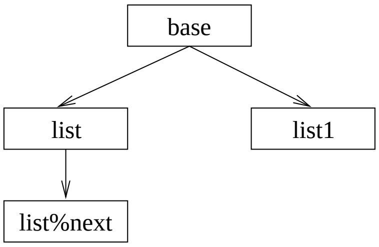


Figure 3.2: SDA Tree

are passed with the standard Fortran 90/HPF semantics (thus with call by reference semantics where possible). As a result of this difference, unsynchronized accesses to variables that are passed as arguments to asynchronously spawned methods may result in a race condition, independent of the intent of the dummy argument. With proper synchronization of asynchronous method invocations by means of *events* such races can be avoided.

Rationale. Call-by-reference semantics conforms to the standard Fortran/HPF calling mechanism and allows in many cases a more efficient implementation of the method invocation by reducing the copy overhead. Moreover, synchronous calls can often be implemented as simple procedure calls which reduces the method invocation overhead significantly.

3.3.2.5 Synchronization

Lightweight SDAs provide the same synchronization facilities as standard SDAs. In particular, their methods may be guarded by a condition clause and asynchronously executing methods may be bound to *events* with which explicit synchronization is possible.

Similarly, only one method of a lightweight SDA may be active unless two methods are explicitly allowed to execute concurrently by means of a *ParBlock*.

3.4 SDA Migration

3.4.1 Introduction

A key issue in any system supporting DHPC applications is the possibility to dynamically re-locate computation to ensure a good load-balancing. Computation re-location may be caused by changes in hardware properties, such as variations in system load in multi-user systems, or by requirements of the algorithm, such as re-distribution of a search tree in branch-and-bound algorithms.

The Opus support for re-locating SDAs is limited. The only possibility of achieving the desired functionality is the exploitation of *persistent* SDAs: an SDA may be saved to persistent storage, destroyed, and eventually re-loaded on different resources. This procedure implies a significant overhead, though.

In this section we present the new intrinsic method **MIGRATE** that allows a more efficient re-location of SDAs.

3.4.2 The Migrate Method

The intrinsic method **MIGRATE** implements the dynamic re-location of SDA objects. It must be accompanied with a resource request indicating the resources to which the SDA should be re-located. An SDA variable to which **MIGRATE** is applied must be bound to an SDA object. Similar to HPF, where an array that may be dynamically redistributed must be annotated with the **DYNAMIC** attribute, SDAs that might be migrated need the Opus **DYNAMIC** attribute. This attribute allows the Opus compiler to generate a more efficient code as will be described in Chapter 5. Moreover, **DYNAMIC** SDAs are accessible by potential runtime schedulers (cf. Chapter 8).

The migration process takes five steps. Let v denote an SDA variable which is bound to an SDA object S allocated on resources *R1*. The execution of a **MIGRATE** method invoked in v with the resource request given by *R2* takes the following steps:

1. Perform resource allocation on *R2*.
2. Allocate an SDA object *S'* on *R2*.
3. Copy all internal SDA data from S to *S'*. This step requires marshaling of dynamic data structures as well as data redistribution.
4. Bind v to *S'*.
5. Deallocate the resources allotted to S.

Lightweight SDAs as described above impose some restriction to the **MIGRATE** method. Since lightweight SDAs use the context of a base SDA, they neither can be freely migrated nor can a base SDA be migrated without taking the migration of all its lightweight children into account:

- A lightweight SDA may only be migrated to resources that are allotted to its base SDA.
- A base SDA may only be migrated if all of its lightweight children are created on exactly the same set of nodes as the base SDA.

Rationale. Lightweight SDAs cannot change their base SDA, therefore they may only migrate within the borders of the base SDA. A migration of a base SDA requires the relocation of all of its children. If a lightweight SDA is assigned only a subset of the nodes allotted to the base SDA, it is in general not possible to translate this subset to the new resources the base SDA is to be migrated.

Example 3.4.1: SDA Migration

The following code excerpt illustrates the migration of an SDA from machine A *to machine* B*. We use the buffer SDA introduced in Chapter 2.*

```
SDA (buffer_type), DYNAMIC :: s

CALL s.CREATE(128) ON (MACHINE='A')
. . .
CALL s.MIGRATE() ON (MACHINE='B')
```

■

The **MIGRATE** method, being an intrinsic method, cannot be contained in a ParBlock directive. Therefore, migration has exclusive access to the SDA which is required in order not to migrate data while some computation is in course.

After the migration has taken place, a stub of the migrated SDA remains on the original resources. This stub is in duty of re-directing any requests that might still arrive on the old location of the SDA. This issue will be discussed in detail in Chapter 5.

3.5 Priority Scheduling

3.5.1 Introduction

With the help of condition clauses the user can influence the scheduling of method executions dependent on the status of the internal data of the SDA. However, a scheduling of method executions *independent* of an SDA's data is not expressible. Such a scheduling possibility gives the user more control over the execution order of SDA methods. In particular, the user might wish that the execution of some critical methods should be given a higher priority than that of others. For instance, the execution of the **MIGRATE** method introduced above should take place immediately independent of any other pending method execution requests.

In this section we present the *priority clause* that can be used to associate a method invocation request with a system dependent priority value.

3.5.2 The Priority Clause

A method activation, irrespective of its synchronousness, may be associated with a *priority clause* that specifies a system dependent *priority value*. The priority value is an integer value in the range between a system dependent minimum and maximum value, which are defined by the Opus constants **MINPRIORITY** and **MAXPRIORITY**, respectively. If no priority clause is specified for a method activation, a system dependent mean value is chosen.

From the set of all *ready* method invocation requests (i.e. all requests whose condition clause is satisfied and whose ParSet is a superset of all active methods) the request with the highest associated priority is chosen to be executed next. These semantics are according to the semantics specified in the pthreads standard [114] for thread scheduling. Similar to the ParBlock directive presented above, the employment of the priority clause may imply fairness problems and even life-locks, though.

Example 3.5.1: Priority Clause

We illustrate the usage of priority clauses with our bounded buffer example in the code fragment below.

```
SDA (buffer_type) buffer
EVENT e
REAL a

. . .
CALL buffer.CREATE(5) ON (MACHINE='A')
. . .
e = SPAWN buffer.getb(a), PRIORITY=MINPRIORITY
. . .
CALL buffer.MIGRATE() ON (MACHINE='B'), PRIORITY=MAXPRIORITY
. . .
WAIT(e)
```

We first create a new buffer on machine `A`*. Method* `getb` *is invoked in the buffer asynchronously with the minimal priority possible. Hence, the method is only served by the buffer if no other ready method with higher priority is available. Then, we migrate the buffer to machine* `B` *giving this request the highest possible priority. This request will be served immediately by the buffer. There is a good chance that* `getb` *has not yet been executed when executing the migration. Hence, the execution of* `getb` *will probably take place already on machine* `B`*.*

■

3.6 Discussion

In this chapter we have presented a set of extensions to the basic Coordination Language Opus. The aim of these extensions is to remove the problems of Opus with respect to DHPC applications which we have identified in Chapter 2. Specifically, we developed new features for expressing more advanced task parallelism and task interaction as well as dynamic task migration means. We summarize these features below, focusing on their support with respect to heterogeneity, parallelism, and communication:

Heterogeneity. Opus components (SDAs) may be created dynamically on resources specified in a resource request. Once created, SDAs may be dynamically re-located by means of the intrinsic method **MIGRATE**. Thus, an Opus program may dynamically react to changes in hardware characteristics, such as system load in multi-user environments and implement dynamic load-balancing strategies which might be required by the application's algorithm. Moreover, SDA migration may be exploited for re-sizing the resources allotted to an SDA.

Parallelism and Communication. Opus provides both, an elaborated data parallel model based on HPF and a flexible task parallel model. While data parallelism is only exploitable within an application component (SDA), task parallelism may be harnessed between as well as within components. In particular, methods of different SDAs may execute concurrently without synchronization since every SDA has its own separate address space. Methods within an SDA may also execute concurrently, however, only if the user guarantees their interference freedom by means of *ParBlock* directives. Moreover, Opus provides flexible means for task scheduling. A *condition clause* may be associated with a method guarding the execution of the method. In addition, a *priority clause* may be associated with a method invocation request enabling priority scheduling of method executions.

Opus components (SDAs) communicate with each other via method invocation. This is a rather expensive and heavyweight process since SDAs have their own context which implies data copies as well as context switches among the different SDA threads. *Lightweight* SDAs reduce this overhead and facilitate standard object composition techniques such as containment and aggregation. In particular, lightweight SDAs re-use the SDA context of a heavyweight base SDA and arguments to their methods are passed with standard Fortran/HPF semantics which is mostly call-by-reference.

Through the introduction of lightweight SDAs Opus became a valuable object-based framework for DHPC applications in that it provides encapsulation features, a hybrid task- and data-parallel model, *active* objects (heavyweight SDAs) for modeling asynchronous tasks, and *passive* objects (lightweight SDAs) for efficient data sharing among tasks.

Chapter 4

OpusJava

4.1 Introduction

We introduced the Coordination Language Opus and its salient features in Chapter 2 and 3. The HPF binding of Opus provides an integrated hybrid parallel model which is indispensable for DHPC applications. Moreover, Opus enhances HPF's language interoperability features by encapsulating HPF programs within re-usable components—SDAs. The SDA mechanism allows different HPF programs to interact in a task parallel manner. By defining Opus bindings to languages others than HPF an application may be comprised of multi-lingual components that interact with each other by means of the well defined SDA model.

In this chapter we develop a Java binding of Opus called *OpusJava* (see also [128, 129]). OpusJava introduces the SDA concept to Java and allows a seamless integration of Java and HPF components within an application. Thereby, OpusJava combines the strengths of the Coordination Language Opus (in particular the integration of HPF style data parallelism with flexible task parallel features) with the strengths of Java (in particular the dynamic networking features) and bridges the gap between efficient high performance computing and dynamic distributed computing. The OpusJava framework consists of two main components, a *Java class library* that introduces the SDA concept to Java, and an *interface* to the Coordination Language Opus, hence to HPF and Fortran 90.

Although other language bindings, such as for C or C++, could have been defined as well, we decided to rely on Java as base language mainly because of its excellent support for heterogeneous platforms and web-computing. Java is, moreover, maturing as a programming language and increasingly employed for high performance applications. See for instance [40, 144, 145] for a discussion of Java prospects in the field of high performance computing. Recently, the JavaGrande forum [120] was established to foster the use of Java in high performance applications and to push the development of new language features that aim to remove the deficiencies of Java with respect to high performance computing.

There are three main scenarios for an employment of OpusJava:

1. Multi-lingual applications comprised of HPF and Java components.

2. As transportation layer within heterogeneous environments.

3. As high level component model for distributed Java applications.

The remainder of this chapter is organized as follows: we illustrate the usage scenarios of OpusJava with a short case study in Section 4.2 before we introduce the OpusJava class library in Section 4.3 and its HPF interface in Section 4.4. A possible integration of high level parallel Java dialects within the OpusJava framework is discussed in Section 4.5. We end this chapter with a summarizing discussion in Section 4.6.

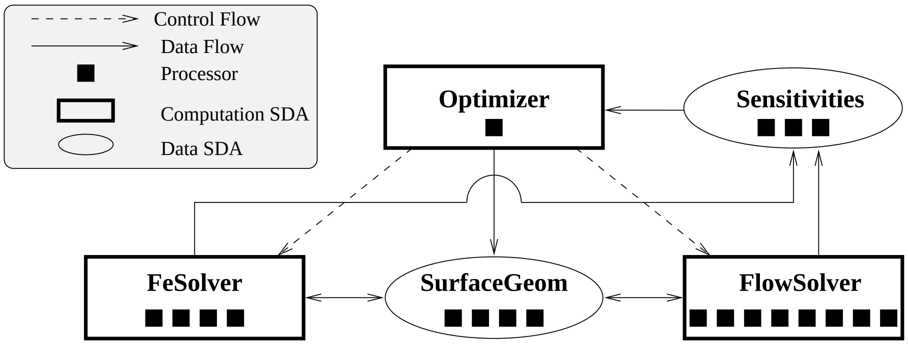


Figure 4.1: A Simple MDO for Aircraft Design [138]

4.2 Case Study: Multidisciplinary Optimization

In this section we illustrate the benefits of OpusJava with a short case study of a typical DHPC application: a multidisciplinary optimization for aircraft design which was described in [138].

The aim of this application is to optimize the design of an aircraft relative to some goal or objective function, such as minimization of gross weight. This is done subject to constraints—specified range and payload, for example. The design cycle starts with these constraints and goals, a base geometry, and initial values for a set of design variables. Each cycle analyzes the current aircraft configuration, as specified by the design variables, to produce a set of output variables. The optimizer evaluates the objective function for this configuration to produce new values of the design variables.

Figure 4.1 shows the structure of an Opus program for an aircraft design application as described above, however simplified in the sense that only two discipline codes, one for the aerodynamic design (SDA `FlowSolver`) and one for the structural design (SDA `FeSolver`) are simultaneously optimized. SDA `Sensitivities` stores sensitivity derivatives, or derivatives of the output variables with respect to the design variables, required by the optimizer. In the figure, rectangles represent SDAs encapsulating computational activities, and ovals represent those encapsulating data repositories. An Opus formulation of this application can be found in [138].

We now discuss the three main usage scenarios of OpusJava with respect to this application:

Multi-lingual applications: Suppose, the optimizer is not a Fortran module as described in the paper but a web-enabled optimizer with a graphical user interface that allows remote monitoring and steering of the optimization process. Such a module would probably be implemented as a Java applet in order to enable a remote, web-based access to the application. With help of OpusJava, the user interface may contain a Java SDA that seamlessly interacts with the HPF computation modules.

Transportation layer: If employed within a complex cluster, the computational modules of the application may be deployed on different supercomputers. As will be described in Chapter 5 Opus requires a common transportation layer that encompasses all machines of a complex cluster. If no higher level communication layer such as MPI is available among the different machines, OpusJava may be employed because of Java's excellent support for heterogeneous systems. Note that in such a scenario the HPF components may even be compiled by different HPF compilers provided that the compiler is supported by Opus. This issue will also be discussed in Chapter 5.

Java component model: Although not the primary use of OpusJava, it is applicable also as a pure Java component framework. In particular, Java components may be coordinated in the same high level manner as HPF components. This allows, for instance, a replacement of the HPF computational modules by Java modules without changing the coordination scheme.

4.3 The OpusJava Class Library

The OpusJava class library is the Java binding of Opus, more specifically it is a framework that introduces the Opus SDA concept with its salient features to Java. While the Coordination Language Opus described in Chapter 2 and 3 is defined as a *language extension* to HPF, OpusJava is a pure Java *class library* without modifications to the language or the JVM. Thus, any existing Java code, even compiled byte-code, can be re-used within an OpusJava application and such applications will run on any standard Java system. This fact favors a wide spread use of OpusJava by making the framework easily available on any system supporting Java.

In the following, we introduce the OpusJava framework, specifically the OpusJava SDA concept, SDA creation, and SDA manipulation, by developing an OpusJava formulation of a bounded buffer used in the standard producer-consumer problem. This problem was also discussed in Chapter 2.

4.3.1 OpusJava ShareD Abstractions

An OpusJava SDA object is, according to standard Opus SDAs, an object encapsulating data and methods. In this respect SDAs are comparable to standard Java objects. However, SDA objects are instantiated on user specified resources and their methods may be invoked by any object, including HPF SDAs, which holds a reference to the SDA. Thus, SDAs can be *remote objects*.

In contrast to Java's remote object model (RMI [175]), OpusJava does require SDA objects neither to implement specific interfaces nor to extend specific classes. Hence, any Java class, even in pre-compiled byte-code representation, may be created as an SDA object. The OpusJava framework provides generic functions for SDA creation and interaction and hides all low level details such as socket communication, RMI, JNI, and synchronization.

Methods of Java SDAs may be invoked in a synchronous or asynchronous manner. Asynchronous method executions are bound to *events* with which explicit synchronization is possible. Method executions may be guarded by *condition clauses* a method may be associated with. While the HPF binding defines new syntax for this purpose (the **WHEN** construct), a condition clause guarding a Java method must fulfill a special naming convention. Specifically, a condition clause must be a logical function, without parameters, whose name is the name of the guarded method postfixed with "`_cond`".

Example 4.3.1: An OpusJava Formulation of a Bounded Buffer

The standard producer-consumer example makes use of a bounded buffer. The code excerpt below illustrates the formulation of such a buffer in Java. Note the two condition clauses `putb_cond` *and* `getb_cond` *which prevent overflows and underflows of the buffer, respectively. An Opus formulation of this example was given in Example 2.3.2*

```
public class Buffer {

   private Integer data[];      // the buffer
   private int size = 0;        // the current size
   private int max;             // the maximum size

   // constructor
   public Buffer(Integer s) {
     max = s.intValue();
     data = new Integer[max];
   }
```

```
    public void putb(Integer i) {
      data[size++] = i;
    }

    public Integer getb() {
      return data[--size];
    }

    // condition clause for putb
    public boolean putb_cond() {
      return (size < max);
    }

    // condition clause for getb
    public boolean getb_cond() {
      return (size > 0);
    }
}
```

As we can see, the declaration of the buffer is pure Java. We only conform to the OpusJava naming convention in defining the condition clauses. This naming convention has only a specific semantics within the OpusJava framework. An object of type `Buffer` *may also be used in any other Java program, in which case* `putb_cond` *and* `getb_cond` *are normal Java methods.*

■

The OpusJava framework provides for mutual exclusive execution of SDA methods, thus ensuring a consistent state of the SDA data. Arguments to Java SDAs are passed with standard RMI semantics; i.e., standard objects are passed by value, remote objects (including SDAs) by reference. The data of SDA objects may only be accessed through the invocation of methods within the object; thus, it is implicitly `private`.

While standard Java RMI does not provide for an interference-free execution of multiple concurrent method invocations within an object and leaves all synchronization burden to the user, OpusJava provides high level synchronization by means of condition clauses and events and furthermore ensures a consistent state of a remote object by imposing mutual exclusive method executions. However, the semantics of these high level features can only be guaranteed within the OpusJava framework. If the user escapes from OpusJava he is solely responsible for proper synchronization. More specifically, OpusJava is a class library that cannot impose nor check specific programming features. Hence, the user of OpusJava is free to employ whatever Java features available also within objects created as an SDA. For instance, an SDA object may register itself as a remote Java object, thus being accessible from outside the OpusJava framework. Multithreading, being an integrated part of Java, may also be exploited within an SDA, giving the user the possibility to create persistent threads that are outside the control of OpusJava. If these features are employed the user is responsible for their safeness, i.e. that they do not interfere with the OpusJava framework. While a safe employment of multiple threads can be guaranteed by assuring that all the threads created in course of an SDA's method execution are terminated before the method returns, accesses to SDA objects from outside OpusJava cannot be guaranteed to work properly in the general case. The user is therefore strongly discouraged to employ these features within an OpusJava program.

By providing all of these features OpusJava enriches the standard Java RMI model with asynchronous method invocation, dynamic object creation on user specified resources, and high level coordination means. These features, which are (at least partially) commonly found in similar Java systems [39, 69, 119, 160], are indispensable for DHPC applications.

OpusJava introduces four new classes visible to the user: class `SDA`, which is a stub for a remote SDA object, class `Event`, which is used to reference asynchronous method invocations, class `Resource` describing an SDA's resources, and class `Request` which is primarily used as wrapper for requests posed to HPF SDAs (cf. Section 4.4). Except of the `Request` class all these classes are `final` classes and thus cannot be extended. The detailed API specification can be found in Appendix C.

4.3.2 SDA Creation

An OpusJava SDA object is created by instantiating a new object of type `SDA`. There are two ways of creating an SDA. First, a new remote SDA object may be instantiated, in which case the constructor has 3 arguments: the name of the class of the remote SDA, an array of input parameters for the remote constructor, and a `Resource` object[1]. Second, a *persistent* SDA which was previously stored to some persistent memory may be *loaded*. In this case, the constructor has other arguments: the file name of the persistent SDA, the code-base, i.e. the location of the file, and a `Resource` object. OpusJava requires all input arguments to constructors as well as to all other methods to be provided as an array of Java objects. Moreover, OpusJava's call-by-value semantics require all arguments to implement Java's `Serializable` interface.

A resource object contains the name of the machine on which the SDA should be allocated (typically an IP-address), and the number of processors to be used. Processors of a multi-processor machine are currently allocated in a system dependent manner. We plan to provide the user with possibilities of extending the resource object allowing a more precise resource request formulation. In particular, means for *indirect* resource allocation (i.e. relative to existing SDAs) will be developed (cf. Chapter 8).

The creation of a new `SDA` object results in the instantiation of a new remote SDA object (i.e. a remote object of user specified type that is under control of OpusJava) on the specified resources. The (local) `SDA` object serves as a stub through which the remote SDA can be accessed.

An object of any Java class may be created as an SDA. There is no need for extending particular OpusJava classes nor for implementing specific interfaces. Java's reflection mechanism is exploited to explore the methods provided by an SDA at runtime. However, OpusJava assumes a specific naming convention for *condition clauses*, as already mentioned above.

OpusJava supports the dynamic class loading features of Java. Hence, the byte-code of SDAs may be dynamically loaded from user specified resources on demand.

Example 4.3.2: Creation of a Bounded Buffer

In order to create an object of type `Buffer` *(cf. Example 4.3.1) as an SDA, we first have to instantiate a new* `Resource` *object:*

```
Resource res = new Resource("milena.par.univie.ac.at",1);
```

In this case, we request one processor of machine `milena.par.univie.ac.at`*. Having created an appropriate resource object, we need to provide the input parameters for the constructor as an array of objects:*

```
Object[] args = new Object[] {new Integer(128)};
```

Now, a new buffer object with a buffer size of 128 can be created as an SDA on machine `milena.par.univie.ac.at`*:*

```
SDA buffer = new SDA("Buffer",args,res);
```

[1] Alternatively, the class name of the remote SDA as well as the input parameters for the constructor may be wrapped by a `Request` object; this possibility is mainly exploited within the context of interfacing HPF SDAs as will be discussed in Section 4.4.

Alternatively, both the class name as well as the input arguments may be enclosed within a `Request` *object:*

```
Request r = new Request("Buffer",args);
SDA buffer = new SDA(r,res);
```

■

4.3.3 SDA Manipulation

The `SDA` stub provides a set of methods to manipulate remote SDA objects (arguments enclosed within `<...>` denote optional arguments):

`Object call(String method, <Object[] args>)` and
`Object call(Request r)` for synchronous method invocation;

`Event spawn(String method, <Object[] args>)` and
`Event spawn(Request r)` for asynchronous method invocation;

`boolean equals(SDA o)` for equality checks among two SDA objects;

`void terminate()` for terminating a remote SDA object;

`void migrate(Resource machine)` for migrating an SDA object;

`void save(String filename, String codebase)` for generating persistent SDAs.

Method Invocation. Two methods are provided for invoking SDA methods in a synchronous (`call`) and asynchronous (`spawn`) way. The arguments of these methods are the SDA method name and an array of input parameters (marshaled as an object array) for the SDA method. Alternatively, these arguments may be wrapped by a `Request` object. Note that all arguments passed to SDA methods have to implement Java's `Serializable` interface.

The `call` method blocks the caller until the results are available. Results of a synchronous method invocation are returned as a Java object exploiting the regular Java result mechanism.

An invocation of the `spawn` method returns an `Event` object immediately and allows the caller to resume its execution. An event is a pointer to an asynchronous method invocation and can be used for synchronization later on. Event objects provide two methods for enquiring the execution status and for synchronizing with asynchronous method executions: `testEvent` and `waitEvent`[2]. The former one immediately returns the current status of the method the event is bound with (i.e. *finished* or *not finished*), while the latter one blocks the caller until the referenced method has finished its execution and the results are available.

The results of an asynchronous method invocation are also returned as a Java object, however, the regular Java result mechanism cannot be exploited because of the asynchronous execution of both the caller and the callee. The event object which is returned as result of executing the `spawn` method is not only used as a synchronization means, but also as a container for outstanding results. In particular, an event object has a private field of type `Object` which is used for storing the method results. The public method `getResults` may be used for obtaining the results from the event. This method is blocking, i.e. it waits until the asynchronous method execution has finished and all of its results are available. Hence, any invocation of `getResults` implicitly synchronizes the caller with the asynchronous method invocation.

[2] We had to modify the Opus syntax (`test` and `wait`), because `wait`, being part of Java's multithreading features, is already defined in any Java object and should not be overwritten.

Events not only provide *data based* synchronization (i.e. implicit synchronization when results of an asynchronous request are being accessed), but also *control based* synchronization that is independent of any data access (by *testing* or *waiting* on an event). In that respect, events are comparable to Multilisp's *Futures* [102], but differ from *Futures* in similar Java systems [39, 119] that only provide data based synchronization.

SDA Equivalence. The `equals` method performs an equality check among two objects of class `SDA`. Specifically, `true` is returned if the argument object refers to the same remote SDA object as the object the method is invoked in; otherwise the result is `false`.

SDA Termination. A `terminate` method for gracefully destroying remote objects is provided as well. Although Java does not provide means for destroying objects and leaves all cleaning mechanisms to the garbage collector, SDAs may be native (HPF) objects and thus support for explicit termination is desirable.

SDA Migration. To support dynamic task re-location, which is a crucial feature for heterogeneous environments, a `migrate` method which receives a new resource object as input parameter is provided. An activation of this method results in a re-location of the SDA object to the new resources. Note that in order to be able to migrate, an SDA has to implement Java's `Serializable` interface.

Persistent SDAs. *Persistent* SDAs are also supported in OpusJava. An SDA may be saved to persistent memory by activating the method `save` which receives a file name as well as a codebase as input parameters. Similar to re-locatable SDAs persistent SDAs have to implement Java's `Serializable` interface.

Example 4.3.3: SDA Manipulation

The code excerpt below illustrates synchronous and asynchronous accesses to the buffer of Example 4.3.1.

```
// input arguments for "putb"
   Object[] inargs = new Object[] {new Integer(4711)};

// synchronous method invocation
   buffer.call("putb",inargs);
// alternative formulation by means of a Request
   Request r = new Request("putb",inargs);
   buffer.call(r);

// synchronous function invocation
   Object result = buffer.call("getb");

// asynchronous method invocation
   Event e = buffer.spawn("getb");
// explicit synchronization
   e.waitEvent();
// use of results
   Object result = e.getResults();

// migration of the buffer
   Resource r = new Resource("sheila.par.univie.ac.at",1);
   buffer.migrate(r);

// save the buffer
   buffer.save("buffer.sda","http://www.par.univie.ac.at/codebase/");
```

■

4.4 Embedding HPF Components into a Java Framework

One of the outstanding features of OpusJava is its interface to the Coordination Language Opus and thereby to HPF and Fortran 90. While HPF provides elaborated downward language interoperability (via the extrinsic mechanism) it is a tedious and difficult task to invoke HPF routines from within other languages. This lack of upward language interoperability is on the one hand due to Fortran 90 itself since Fortran 90 does not define interfaces to any other language; on the other hand interoperability is also hampered by HPF since there is no standardization of HPF's internal representation, in particular of processor arrays and distributed data.

Opus, with its binding to HPF and Java, offers a common infrastructure and abstraction that allows an intermixing of Java and HPF components. More specifically, the OpusJava class library introduced above can be employed not only to create and manipulate Java SDAs but also HPF SDA. Similarly, HPF SDAs may create and manipulate Java SDAs, as well. Moreover, references to SDAs of any language may be passed as method arguments thus crossing the language borders.

4.4.1 Fortran Interoperability

For the time being, Fortran does not define interfaces to any other language although most Fortran compiler vendors provide their own in-house interfaces to other languages, in particular C. Within the current standardization effort for Fortran 200x a Fortran-C interface will be defined [118]. Java, on the other hand, has defined a sophisticated interface to C, the *Java Native Interface (JNI)* [173]. Therefore, interoperability between Fortran and Java is best accomplished via C. More specifically, any data passed from Fortran to Java is converted to an appropriate C data type according to the type mapping defined by the employed compiler or by the upcoming standard and eventually converted to Java according to the JNI specifications.

The conversion of non-basic data types from C to Java and vice versa is however a non-trivial task. In particular, multidimensional arrays have different representations in C and Java. While C arrays represent a contiguous, rectangular memory area (as do Fortran arrays), Java represents multidimensional arrays as "arrays of arrays" which are linked by pointers. Therefore, expensive argument marshaling and unmarshaling is required. However, there are attempts to enrich Java with a viable array concept [42, 145] which would greatly reduce this overhead.

The task of data marshaling is taken care of by the Opus compiler which generates appropriate C functions for translating Fortran data to Java as will be described in Chapter 5. The user is relieved from this task, though he has to obey the different memory organization in C and Fortran (row-major vs. column-major order).

4.4.2 HPF Interoperability

Making HPF interoperable with Java is an even more demanding task than Fortran interoperability since HPF does not define an upward interface that allows HPF routines to be invoked from a language other than HPF. Specifically, calling conventions with distributed arguments are not specified in HPF.

Opus circumvents this problem by providing a Fortran *gateway* for interaction between Java and HPF. More specifically, any Opus/HPF application contains an *External SDA Stub* which is responsible for routing requests from and to OpusJava applications. This Stub which is generated by the Opus compiler (cf. Chapter 5) acts as a multiplexer that transfers OpusJava requests marshaled by C wrappers to the respective HPF SDA and vice versa. Hence, there is no direct interaction between Java and HPF. Distributed data is handled by appropriate gather/scatter operations executed within the Stub. This issue will also be described in detail in Chapter 5.

Figure 4.2 illustrates the components involved in interfacing Opus/HPF with OpusJava: a request from a Java SDA is marshaled by C wrappers and transferred to the External SDA Stub which routes the request to the appropriate HPF SDA taking care of the required data distribution. Similarly, an HPF SDA may interact with a Java SDA by sending a request to the Stub which

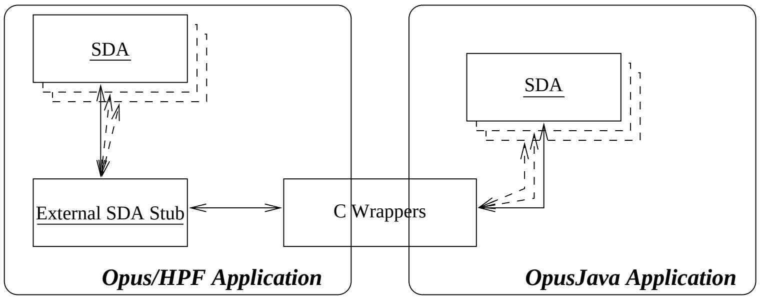


Figure 4.2: Components of the Opus HPF/Java Interface

collects all the required distributed data. The request is translated to Java in the C wrappers and forwarded to the appropriate Java SDA.

In the following we take a closer look at the issue of translating arguments between HPF and Java. As already mentioned, the data distribution issue is taken care of by the External SDA Stub, hence the problem is reduced to argument transfer between Fortran and Java. Note that due to the not standardized Fortran/C interface only basic data types respectively arrays of basic data types may be transferred to and from Java methods. More complex structures will become possible once a mapping between C structures and Fortran 90 derived types will be established in the upcoming Fortran 200x standard [118].

Calling HPF from Java. As already discussed above, OpusJava provides two ways of creating SDAs resp. invoking methods. Either, the name of the SDA/method may be explicitly passed together with the input arguments marshaled as an array of objects, or a `Request` object may wrap this information. Such `Request` classes are exploited for manipulating HPF SDAs in that the Opus compiler generates appropriate Java classes extending `Request` for all HPF SDAs and their methods. These classes are kind of a Java interface to the respective HPF routines. By providing these classes the user is relieved from argument marshaling and, moreover, a static validation of method signatures is possible. Specifically, for every HPF SDA type the Opus compiler generates a class extending `Request` named with the name of the SDA type and a trailing "`_request`"; for all methods of an SDA the respective class name is composed of the SDA type name concatenated with the respective method name and a trailing "`_request`". These classes are accessible via the `OpusJavaSupport` package. In addition, Java classes that encompass all output parameters of HPF methods, as well as potential results in case of a function method, are defined. The name of such a class is composed of a leading "`Result_`" followed by the SDA type name concatenated with the respective method name. `Result` classes are passed to the OpusJava framework as components of `Request` objects such that the framework is able to marshal the results of a method invocation appropriately; otherwise, the results are marshaled within an `Object` and the user has to unmarshal them explicitly. Result classes are also put into the `OpusJavaSupport` package.

Example 4.4.1: Java-HPF Interoperability

To illustrate the interoperability features of OpusJava, we present an OpusJava program that creates and accesses an HPF buffer. Recall the HPF formulation of our bounded buffer from Example 2.3.2:

```
     SDA TYPE buffer_type(size)
!HPF$ PROCESSORS directives
      INTEGER :: size
      REAL :: fifo(0:size-1)
      INTEGER :: count=0
      INTEGER :: px=0, gx=0
!HPF$ ALIGN & DISTRIBUTE directives
      . . .
     CONTAINS
      SUBROUTINE putb(x) WHEN (count .LT. size)
        REAL, INTENT(IN) :: x
        fifo(px) = x
        px = MOD(px+1,size)
        count = count + 1
      END

      SUBROUTINE getb(x) WHEN (count .GT. 0)
         REAL, INTENT(OUT) :: x
         x = fifo(gx)
         gx = MOD(gx+1,size)
         count = count - 1
      END
      . . .
     END SDA TYPE buffer_type
```

From the SDA definition above the Opus compiler generates the following `Request` *classes as well as the result class for subroutine* `getb`*:*

```
// SDA buffer_type
package OpusJavaSupport;
public class buffer_type_request extends Request {
   public buffer_type_request(int size) {
      super("buffer_type", new Object[] {new Integer(size)});
   }
}

// method putb
package OpusJavaSupport;
public class buffer_type_putb_request extends Request {
   public buffer_type_putb_request(float x) {
      super("putb", new Object[] {new Float(x)});
   }
}

// results of method getb
package OpusJavaSupport;
public class Result_buffer_type_getb {
   public float x;
}
```

```
// method getb
package OpusJavaSupport;
public class buffer_type_getb_request extends Request {
   public buffer_type_getb_request() {
      super("getb",Class.forName("Result_buffer_type_getb"));
   }
}
```

An OpusJava program may create and access an object of SDA type `buffer_type` *as follows:*

```
   import OpusJavaSupport.*;

// generation of a new Resource object
   Resource res = new Resource("milena.par.univie.ac.at",1);

// creation of a "buffer_type" SDA with size 128
   SDA buffer = new SDA(new buffer_type_request(128), res);

// synchronous invocation of "putb"
   buffer.call(new buffer_type_putb_request(47.11));

// synchronous invocation of "getb"
   Result_buffer_type_getb result =
                 (Result_buffer_type_getb) buffer.call(
                                new buffer_type_getb_request());

// asynchronous  invocation of "getb"
   Event e = buffer.spawn(new buffer_type_getb_request());

// explicit synchronization
   e.waitEvent();

// use of results
   Result_buffer_type_getb result =
                         (Result_buffer_type_getb) e.getResults();
```

Note that the OpusJava code above is almost exactly the same as the one presented in Example 4.3.2 and 4.3.3. The only difference is that the compiler generated `Request` *classes are exploited for indicating the respective SDA and method names and for marshaling of input arguments instead of providing this information explicitly. In other words, neither on the HPF side nor on the Java side special provisions need to be made for language interoperability.*

■

Calling Java from HPF. As will be discussed in Chapter 5 the Opus compiler requires interfaces of all the SDAs (hence also Java SDAs) in a program at compile time. For this purpose, *Fortran SDA interfaces*, designed along the lines of Fortran 90 procedure interfaces [116], have to be provided by the user. These interfaces are the Fortran counterparts of the Java request classes. With help of SDA interfaces the Opus compiler is able to generate all necessary components that allow an HPF SDA to interact with OpusJava SDAs. Specifically, an SDA interface consists of the keyword **SDA INTERFACE** followed by the SDA type name, the interface body which conforms to the Fortran 90 interface body, and the closing keyword **END SDA INTERFACE** (cf.

Appendix A). Although SDA interfaces currently have to be provided by the user, their generation could quite easily be automated by means of a simple Java pre-processor. The development of such a pre-processor is planned in the near future.

Example 4.4.2: HPF-Java Interoperability

Recall the Java class `Buffer` *of Example 4.3.1. Its corresponding Fortran SDA interface has the following form:*

```
SDA INTERFACE Buffer
   ! constructor
   SUBROUTINE Buffer(s)
      INTEGER, INTENT(IN) :: s
   END SUBROUTINE Buffer

   ! methods
   SUBROUTINE putb(i)
      INTEGER, INTENT(IN) :: i
   END SUBROUTINE putb

   INTEGER FUNCTION getb()
   END FUNCTION getb
END SDA INTERFACE Buffer
```

Note that the interface does not contain condition clauses, since they are only relevant to the SDA object itself and not to any other object having a reference to the SDA object. Given this interface definition an Opus program may create and access Java objects of type `Buffer` *just as any other HPF SDA:*

```
. . .
SDA (Buffer) :: s
INTEGER i

CALL s.CREATE(128) ON (...)

CALL s.putb(5)

i = s.getb()
. . .
```

Through its declaration within an `SDA INTERFACE` *SDA* `Buffer` *is recognized as an external SDA by the Opus compiler. Consequently, requests to SDAs of this type are routed through the External SDA Stub.*

■

4.5 Hybrid Parallelism in OpusJava

In this chapter we have so far presented our design for the Java binding of Opus and its interface to the HPF binding. The main goal of Opus is the provision of a high level component framework that integrates coarse grained task parallelism with fine grained data parallelism. This goal is achieved in the Coordination Language Opus by augmenting an existing data parallel language (HPF) with task parallel features. OpusJava, instead, is a purely task parallel framework. Although methods of Java SDAs may well exploit parallelism, for instance by means of the Java threading features, this internal parallelism is not well integrated into the OpusJava framework, difficult to use, and error-prone, as mentioned above.

To enrich OpusJava with a viable hybrid parallel model we would require a high level, fine grained parallel Java dialect, much like HPF is to Fortran. Such a dialect could then be used as building block of Java SDAs. Although there is currently no consensus on parallel Java dialects many approaches exist which intend to include high level (data) parallel features in Java [39, 41, 60, 112, 119, 125, 160, 182] (see also Chapter 7). In this section we review two of these approaches, HPJava [41, 42] and SPAR [182], and explore their potential for supporting hybrid parallelism in OpusJava.

4.5.1 High Level Parallelism in Java

HPJava. HPJava [41, 42], under development at the University of Syracuse, extends the Java language with some syntax for manipulating distributed arrays. Arrays in HPJava are real multi-dimensional arrays and not just arrays of arrays as in Java. They are distributed onto processors that are represented as *processor groups* using some predefined distribution strategy, such as *block* or *cyclic*. Moreover, a parallel loop construct, the *overall* construct is defined. HPJava has been inspired by HPF and it took over many of the concepts introduced in HPF. In particular, these concepts include processor objects, distributed arrays (including the possibility of collapsing or replicating array dimensions), and *ghost regions*, similar to HPF's *shadow region*. However, while the HPF philosophy is to exploit advanced compilation technology to generate an explicit SPMD program, HPJava relies on library functions which the user has to employ for accessing non-local data. These library functions are mostly implemented in a general communication library that was originally designed to support HPF compilers, adlib [43]. It is the major design philosophy of HPJava to make the user aware of communication, thus forcing him to spend more effort in searching for algorithmic solutions that might require less communication and might therefore be more efficient. In this respect, HPJava's philosophy is similar to that of Co-Array Fortran [149] which also introduces specific syntax for accessing non-local data.

Example 4.5.1: HPJava: Matrix Multiply

We illustrate HPJava with a standard matrix-matrix multiply example:

```
Procs2 p = new Procs2(P,P);

on(p) {
   Range x = new BlockRange(N, p.dim(0));
   Range y = new BlockRange(N, p.dim(1));

   float [[,]] c = new float [[x,y]];
   float [[,*]] a = new float [[x,N]];
   float [[*,]] b = new float [[N,y]];

   // initialize a and b

   overall(i = x for :) {
      overall(j = y for :) {
         float sum =0;
         for(int k = 0; k<N; k++)
            sum += a[i,k] * b[k,j];
         c[i,j] = sum;
      }
   }
}
```

In this code a 2 dimensional processor group with `P` *processors in each dimension has been created. On this group (*`p`*) 2 distribution ranges,* `x` *and* `y`*, are defined which distribute an index space of* `N` *elements in a block-wise fashion along one dimension of* `p`*. With help of these ranges the three matrices are allocated and distributed:* `c` *is distributed in equally sized blocks along both dimensions while* `a` *and* `b` *are replicated along one dimension. Thus all data needed for the inner scalar product are already in place for the computation—no communication is needed.*

■

SPAR. An approach different to HPJava is taken in the SPAR [182] project at the Technical University of Delft. While HPJava supports explicit data parallelism and requires the user to take care of non-local data accesses, SPAR introduces medium-grained task parallelism where the mapping of tasks and data is executed by the compiler and runtime system transparent to the user. More specifically, SPAR is a superset of Java providing a parallel *foreach* loop, parallel code blocks (the *each* construct), and extensive support for arrays (including truly multidimensional arrays and *elastic* arrays that are able to grow and shrink dynamically).

A SPAR program is compiled to an intermediate language "Vnus" [181]. This intermediate language is eventually compiled to C++ code with explicit parallelization and communication. Performance prediction techniques are exploited for automatically mapping parallel task to the target architecture.

Example 4.5.2: SPAR: Matrix Multiply

Similar to Example 4.5.1, where we discussed an HPJava formulation of a standard matrix-matrix multiply example, we illustrate SPAR with the same example:

```
float a[*,*] = new float[N,N];
float b[*,*] = new float[N,N];
float c[*,*] = new float[N,N];

// initialize a and b

foreach (int i = 0:(N-1)) {
   foreach (int j = 0:(N-1)) {
      float sum = 0;
      for(int k = 0; k<N; k++)
            sum += a[i,k] * b[k,j];
      c[i,j] = sum;
   }
}
```

As can be seen from the code excerpt above, the SPAR code for matrix-matrix multiply is very similar to the HPJava code presented in Example 4.5.1, except that there is no explicit data mapping in SPAR.

■

4.5.2 Embedding Parallel Java Components in OpusJava

We briefly discussed two high level parallel Java dialects, HPJava and SPAR, above. Both systems can be considered as a practical complement to OpusJava since both of them provide fine grained loop parallelism which could be exploited within coarse grained OpusJava SDAs. In the following, we take a closer look at some technical issues that would raise in the OpusJava context.

First of all, both of these systems are at an experimental stage and not yet fully specified and implemented, thus we can only discuss potential problems and benefits.

An integration of HPJava and OpusJava seems to be straight-forward since both systems rely on Java itself. An SDA object may define distributed data as well as parallel constructs within its methods. Of course, the OpusJava framework needs to be aware of parallel HPJava objects which could be accomplished by appropriate HPJava processor objects that are handled within the framework. Problems arise in the context of distributed method arguments, though. In particular, the user is required to deal with data locality explicitly in HPJava and from the current HPJava definition it is not clear how distributed input arguments to methods are being handled.

A fine grained parallel formulation of SDAs by means of SPAR would require similar means as the Opus Java/HPF interface described above since SPAR is compiled to explicitly parallel C++ code. In particular, extensive modifications of the SPAR and/or the underlying Vnus compiler would be required. However, given the possibility of these modifications, an integration of SPAR could be realized quite efficiently and, equally important, without that many user interactions that would be required by HPJava.

In summary, the integration of high level parallel Java modules within OpusJava seem to be promising and realizable along the lines of the HPF integration. However, real integration tests have to be postponed until stable systems are available.

4.6 Discussion

Similar to the preceding Chapters 2 and 3 we end this chapter with a summarizing discussion of the features of OpusJava, specifically focusing on its support for DHPC application. In doing so, we review the features of OpusJava by means of the criteria presented in Chapter 1.

Heterogeneity. OpusJava deals with heterogeneity issues in several aspects. First, OpusJava is a pure Java framework, thus it benefits from Java's support for heterogeneous systems.

Second, Java's already quite elaborated heterogeneity support is extended with respect to dynamic task management. In particular, OpusJava allows remote objects (SDAs) to be dynamically created on user specified resources as well as to be dynamically migrated during program execution whereas Java's remote objects cannot be migrated.

Third, through its interface to the Coordination Language Opus OpusJava enhances the support of Opus for heterogeneous architectures in that OpusJava may interconnect Opus modules instantiated on different, heterogeneous, machines. This issue will be further discussed in Chapter 5.

Interoperability. As we have discussed in Chapter 2, the Coordination Language Opus enhances HPF's vertical interoperability features (primarily the **EXTRINSIC** mechanism) with horizontal interoperability features. These features are further strengthened in OpusJava that defines an interface to the Coordination Language Opus and thus allows a seamless integration of HPF and Java modules. It is worth noting that OpusJava does not impose a particular runtime support for these HPF modules—this is in contrast to other component frameworks that integrate multiple languages such as Legion [92, 93] or Globus [78] which require runtime support systems to build upon, or to be emulated by, their own systems. Thereby, even different HPF compilation systems may be seamlessly integrated within an OpusJava application. This feature allows in particular to leverage mature HPF systems on any machine of a complex cluster. Furthermore, Java's JNI mechanism allows to escape from OpusJava and to integrate native methods within OpusJava objects.

Parallelism and Communication. OpusJava provides a high level component framework that supports the coordination of concurrently executing objects. In this respect, OpusJava is similar to Java's Remote Method Invocation (RMI) mechanism, however at a higher level and with more advanced features such as synchronous as well as asynchronous method invocation and object migration. High level synchronization means are provided, as well. These include condition clauses to guard a method's execution and event objects for synchronization with asynchronously executing methods.

Apart from these coarse grained task parallel features fine grained parallelism may be exploited by employing Java's multithreading features. It is however a difficult task to do ones own multithreading that does not interfere with the OpusJava mechanism as already discussed in Section 4.3.1. Therefore, higher level approaches for parallel Java objects have been surveyed and strategies for their integration within the OpusJava framework have been sketched in Section 4.5.

Similar to RMI, OpusJava objects communicate with each other via method invocation, the user has to provide input arguments marshaled as an object array, though. For interoperability with HPF SDAs the Opus compiler generates supporting classes that provide an interface to HPF SDAs and help in appropriately marshaling arguments to their constructors as well as methods.

Summary. OpusJava is a thin framework for DHPC that provides high level communication and coordination means and allows a seamless and efficient integration of high performance HPF components into larger, distributed, Java based environments. OpusJava hides all low level details encountered in distributed computing such as communication and data format conversion. Moreover, OpusJava employs Java's dynamic class loading features which allow new modules to be plugged into a running application dynamically.

OpusJava is a pure class library without modifications of the Java language or the JVM. This fact, however, has the drawback that Java's reflection mechanism is exploited to check the validity of OpusJava requests at runtime. In particular, the generic OpusJava interfaces do not allow static type checking nor checking of method signatures within SDAs. These features could be integrated within OpusJava by means of an IDL, however at the expense of additional compilation effort. Moreover, the requirement for IDLs would render the creation of dynamically loaded byte-code classes as SDAs impossible.

Currently, OpusJava supports all features defined for the basic Coordination Language Opus (cf. Chapter 2). These features include *synchronous* and *asynchronous method invocation*, synchronization via *condition clauses* and *events*, and *persistent* SDAs. From the Advanced Opus Features presented in Chapter 3, only SDA *migration* is supported, so far. We consider SDA migration as one of the crucial features that need to be available in any viable DHPC system. All other features defined in Chapter 3 are improvements to the language and/or parallelism model of the Coordination Language Opus. We plan to include them in future versions of the OpusJava framework.

Chapter 5

Design of an Opus Implementation

5.1 Introduction

In the previous chapters we have presented the Opus system, in particular we defined the Coordination Language Opus and the OpusJava framework. In this chapter we describe the design of a system that efficiently implements the Opus constructs.

As general guidelines for our system design we identify a number of properties the design has to fulfill:

Portability: The Opus system should be easily portable across a wide range of systems.

Efficiency: Being a system for distributed high performance computing, efficiency is of paramount importance. Not only the Opus system itself should be implemented as efficiently as possible, it should also allow the exploitation of the most efficient supporting systems (such as message passing or numerical libraries) available on any given machine.

Extendibility: Our design should be open to further developments of the language and/or supporting systems.

Software re-use: In a complex system as Opus, software re-use is of crucial importance. This holds in particular for the data parallel parts of Opus.

A modular design of the Opus system allows the incorporation of new features as well as the replacement of existing sub-systems with a justifiable effort. Hence, we designed the Opus system as a layered system comprised of:

- Opus independent runtime support systems, such as message passing libraries, multithreading libraries, etc;
- Opus dependent runtime systems; and
- The Opus compiler.

Figure 5.1 illustrates the system layers for both, the Opus language and OpusJava. One of the major design decisions for the Opus language was where to put the borderline between the compiler and the runtime system. As we will explain below in more detail we decided to make use of only a thin language dependent runtime system and used the compiler in conjunction with pre-defined compilation templates to implement most of the Opus semantics. However, we were able to implement OpusJava as a pure class library without any compiler support by exploiting advanced Java features such as the reflection mechanism.

The remainder of this chapter is organized as follows: In Section 5.2 we discuss the implementation of the Opus language covering all system components, ranging from the low level virtual machine up to the high level compiler component. Special issues that arise in the context of the Advanced Opus Features are tackled as well. In Section 5.3 we present the design of the OpusJava

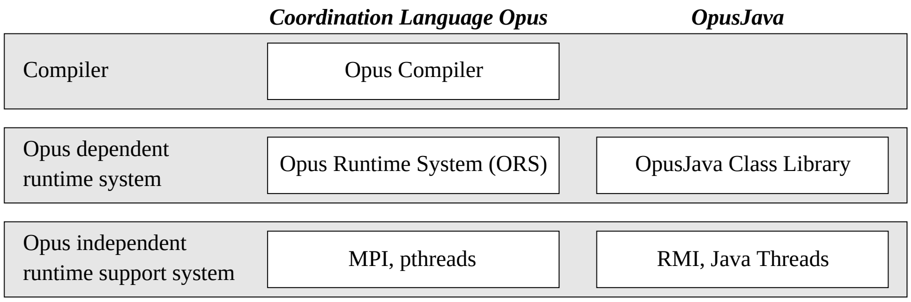


Figure 5.1: Layers of the Opus System

class library and its interface to the Opus language. We end this chapter with a summarizing discussion in Section 5.4. In describing our system design we presume some basic knowledge of Fortran 90, HPF, and Java.

5.2 Implementing the Coordination Language Opus

5.2.1 Introduction

In this section we describe the design of a system that efficiently implements the Coordination Language Opus (or simply Opus in the following), i.e. the HPF binding of Opus (see also [130, 131]). We provide an overview of the whole system, ranging from the basic supporting systems up to the high level compiler component. We focus specifically on the task parallel features of Opus. The data parallel (HPF) portion is left to be handled by an existing HPF compilation system (such as the *Vienna Fortran Compiler VFC* [21]).

The transformation of an Opus source code into an executable program is a 3-stage process as can be seen in Figure 5.2:

- 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.
- This HPF 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.

Although we have designed our Opus system to work in close cooperation with a specific HPF compiler (VFC) there is nothing inherently in our design which prohibits the integration of HPF compilers others than VFC. We will touch this issue in Section 5.2.5.

In describing our implementation design we take a bottom-up approach starting with a discussion of the Opus Runtime Systems and its supporting components in Section 5.2.2 followed by a presentation of the code translation process performed by the Opus compiler in Section 5.2.3. Issues in distributed SDAs, in particular the data exchange mechanisms, are discussed in Section 5.2.4. In Sections 5.2.5 and 5.2.6 we give some practical remarks on our implementation design, discuss design alternatives together with their advantages and disadvantages, and the motivation for our choices. Finally, issues specific to the Advanced Features presented in Chapter 3 are discussed in Section 5.2.7.

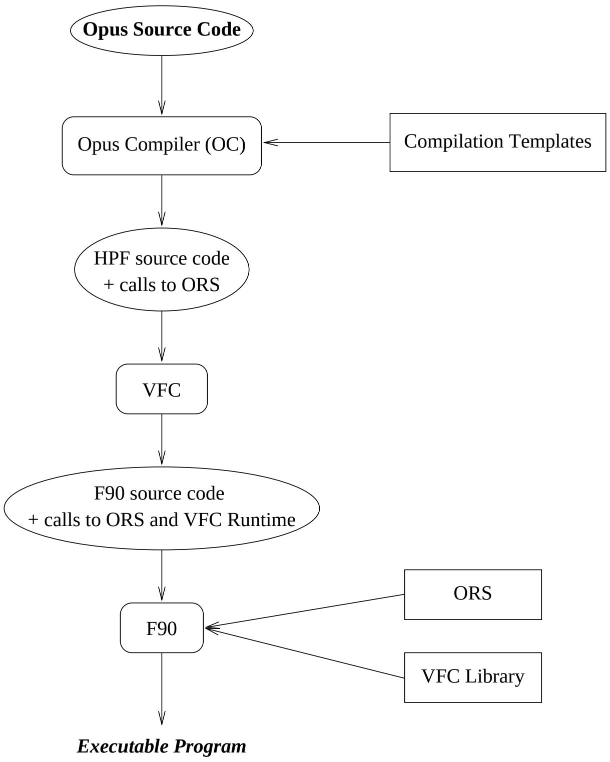


Figure 5.2: The Opus Compilation Process

5.2.2 The Opus Runtime System

The Opus Runtime System (ORS) provides an interface to the underlying system components, such as threading and communication systems, and 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 method invocation (MI) handling. Moreover, the ORS facilitates task synchronization via the event mechanism.

In the following, we take a closer look at the supporting system components and the runtime support for method invocation before we present the most important ORS routines in detail.

5.2.2.1 Supporting System Components

Recall the components of the Opus execution model (cf. Figure 5.3 that was also discussed in Chapter 2): each SDA is associated with a unique *context* that is established on one or several *nodes*. Within a context several *SDA threads* exist.

The ORS provides the basic mechanisms for establishing (distributed) SDA contexts and for interaction (communication) among such contexts. For these tasks language independent runtime support systems are required:

SDA Interaction: Since SDAs may reside on different sets of nodes and nodes do not necessarily physically share memory we require communication mechanisms that do not assume a shared memory model for realizing SDA interaction; for instance standard message passing libraries.

SDA Context: The establishment of an SDA context could be realized by mapping an SDA context to a set of processes (one process per node), but this approach suffers from two major drawbacks: the inability to control scheduling and the costly context switching

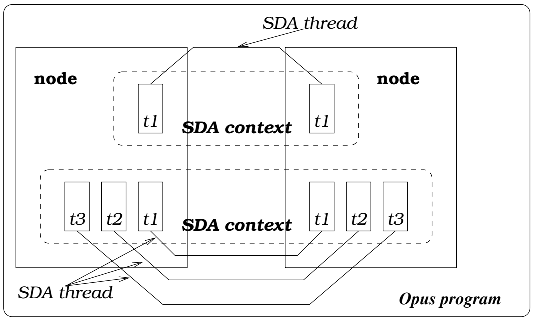


Figure 5.3: Opus Execution Model Components Revisited

overheads (cf. [100]). In light of the problems with using a process-based model, we have elected to use a *multithreading* model where an SDA context is realized by a set of lightweight, user-level threads. However, some support for a process-based model is necessary for platforms which do not allow multithreading as will be discussed in Section 5.2.5.

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 [141], TCP/IP [113], and POSIX threads [114] (cf. Figure 5.4). These low level components can be seen as part of the virtual machine on top of which an Opus application will be executed. The following discussion is, without loss of generality, based upon the assumption that MPI and pthreads are exploited.

We have taken a simplified approach to node management: we presume that all the required nodes are assigned to an Opus application at program start and remain invariant during the lifetime of the program. SDA contexts may be created and destroyed dynamically on any subset of these nodes. The extension of our design to support dynamic acquisition of new nodes is subject to ongoing work.

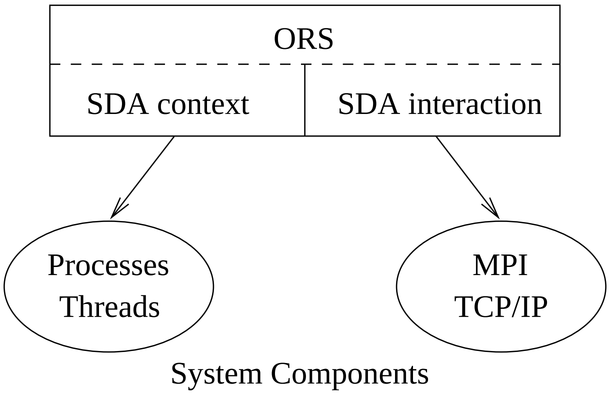


Figure 5.4: The Opus Runtime System and its Supporting System Components

```
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
   TYPE(ors_distribution_descriptors), POINTER :: caller_descriptors
   TYPE(ors_distribution_descriptors), POINTER :: callee_descriptors
END TYPE ors_execution_rec
```

Figure 5.5: The Execution Record

5.2.2.2 ORS Support for Method Invocation

Method invocation requests (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 together with their HPF distribution descriptors used in both the caller and the callee. Figure 5.5 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, type `ors_generic_type` is a compiler generated abstraction of the method arguments, and `ors_distribution_descriptors` holds all of the HPF distribution descriptors of the 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 5.2.3).

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 [43, 59, 162] for this purpose (cf. Section 5.2.4). In implementing the ORS, we were careful to avoid the underlying message passing system when communicating between two SDAs co-located on the same nodes, and used direct memory copies in these situations.

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 storing requests in the order of their arrival such that always the oldest request with a satisfied condition clause is chosen for execution.

5.2.2.3 Main ORS Routines

In the following we provide a more detailed discussion of the main ORS routines whose interface is given in Figure 5.6. 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 5.2.6.

- `ors_initialize_sda(iq,oq)`: The initialization routine of an SDA. It allocates the *inqueue* `iq` and *outqueue* `oq` and constructs a new communicator containing all the nodes on which the SDA has been created. This communicator is provided to the data-parallel VFC-runtime system which will use this communicator for its own communication needs.

```
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_mi_queue) queue]
                             TYPE(ors_execution_rec), pointer :: return_rec
ors_insert_rec(r, queue)    [TYPE(ors_execution_rec), pointer :: r
                             TYPE(ors_mi_queue) queue]
ors_delete_rec(r, queue)    [TYPE(ors_execution_rec), pointer :: r
                             TYPE(ors_mi_queue) queue]

ors_new_event(e,r)          [TYPE(ors_event), pointer:: e
                             TYPE(ors_execution_rec) r]
ors_test(e,s) result(finished)
                            [TYPE(ors_event) e
                             TYPE(ors_sda_handle) s
                             LOGICAL finished]
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_receive_mi(r)           [TYPE(ors_execution_rec), pointer :: r]
ors_send_synchronous_mi(r)  [TYPE(ors_execution_rec) r]
ors_send_asynchronous_mi(r) [TYPE(ors_execution_rec) r]
ors_send_exec_ack(r)        [TYPE(ors_execution_rec) r]
ors_handle_return(r)        [TYPE(ors_execution_rec) r]
```

Figure 5.6: Main ORS Routines

- `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`). After that it is safe to delete the SDA and to return.
- `ors_get_rec(r, queue) result(return_rec)`: Depending on whether the optional argument `r` is available, a pointer to the oldest execution record ready for processing or a pointer to the record matching `r` is returned. The latter case arises if the execution progress of an MI is queried by the intrinsic "wait" or "test" procedures. 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 implements 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 implements 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 routine 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_receive_mi(r)`: This routine implements the process of receiving MIs. It is 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 5.2.4 for a detailed discussion of input argument transfers.

- `orts_send_exec_ack(r)`: This routine returns an execution acknowledgment for request `r` to the caller of `r`. Note that similar to the sending routines no results are communicated by this routine but rather by compiler generated data redistribution mechanisms (see Section 5.2.4).

- `ors_handle_return(r)`: This routine is invoked on receipt of 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) about the termination of the MI. 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.

5.2.3 The Opus Compiler

The Opus compiler is a source to source translator that restructures the Opus code and produces blocks of HPF code that utilize the functionalities provided by the ORS. More specifically, the compiler translates the code of an SDA into a set of HPF 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.

As discussed in Section 5.2.2, the mapping of SDAs 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 5.2.5. Hence, an SDA thread is realized by a (distributed) lightweight, user level thread. Recall that an SDA thread is required to be able to receive MIs independent of its state (cf. Section 2.2). Therefore an SDA thread requires at least two (distributed) lightweight threads, one for receiving MIs, another for managing all the required actions with respect to method execution. Within a node lightweight threads building an SDA thread communicate via a shared memory segment.

As shown in Figure 5.7 the three components of an SDA thread are:

- A shared memory area, in which the MI-queues as discussed in Section 5.2.2 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 associated condition clauses and executes the respective methods.

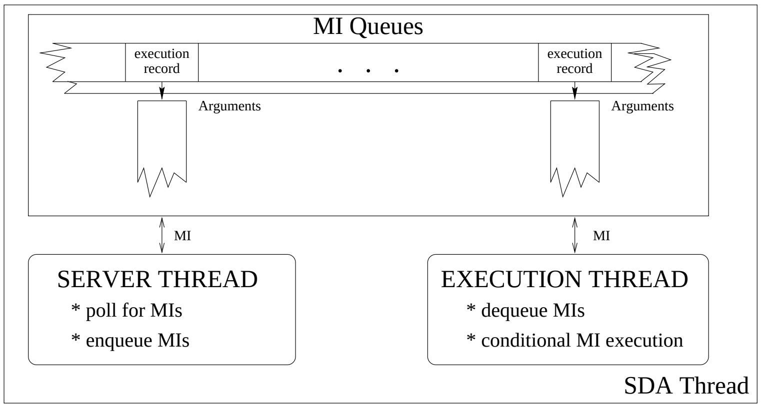


Figure 5.7: Structure of an SDA Thread

Both threads execute asynchronously in parallel; the only synchronization required is a mutually exclusive access to the shared queues. Note that in case of distributed SDAs a parameterized instance of these three components is set up an every node allocated to an SDA context.

In the following, we describe the compilation of SDAs, including creation and method invocation, and introduce the associated compilation templates. For the sake of clarity we restrict our discussion mainly to non-distributed SDAs that are instantiated on a single node. Issues specific to distributed SDAs will be elaborated in Section 5.2.4.

5.2.3.1 SDA Creation

At program start a special thread, called the *creation thread*, is spawned on every node allocated to the program. When a new SDA needs to be *created*, the caller SDA (or the main procedure) sends a message to the creation threads of all the nodes on which the SDA is to be established. If those nodes are not a subset of the nodes available to the program, a runtime exception will be raised. In response to this message, the creation threads establish the new SDA context.

Recall from Chapter 2 that the creation process of an SDA takes five steps:

1. Determine an SDA type, t.
2. Perform resource allocation, as specified in the (optional) resource request.
3. Allocate data structures for B, as determined by t, the resource allocation, and distribution and alignment directives associated with t.
4. Initialize B.
5. Bind B to v.

Figure 5.8 illustrates the compilation template for a creation thread. After it is spawned, it blocks in a call to `ors_receive_mi`. Apart from a termination signal a creation thread may only receive a creation request. The `method_id` component of the execution record of such requests is used to specify the type of the SDA to be created. Depending on the respective type, actual arguments for all dummy arguments are received (step 1. of above). Then, both the respective server and execution threads are spawned (step 2.). As will be discussed below, the execution thread allocates the shared memory area (the inqueue and outqueue) as well as all the internal data structures (step 3.). The (optional) constructor routine (step 4.) provided for initialization purposes is also called within the execution thread. During this initialization process the creation thread is blocked waiting on a condition variable until it is signaled by the execution thread that

```
SUBROUTINE creation_thread
  TYPE(ors_execution_rec), POINTER :: b

   DO ! forever
      CALL ors_receive_mi(b) ! blocking

      SELECT CASE (b%method_id) ! method_id refers to the SDA type
      CASE(1)
         ! receive arguments for SDA type 1
         ! create the server thread
         CALL ors_thread_create(<sda_name_server>)
         ! create the execution thread
         CALL ors_thread_create(<sda_name>)
         CALL ors_wait_creation_finished
      ...
      CASE(n)
         ! receive arguments for SDA type n
         CALL ors_thread_create(<sda_name_server>)
         CALL ors_thread_create(<sda_name>)
         CALL ors_wait_creation_finished
      CASE (termination_id)
         EXIT
      END SELECT
      CALL ors_send_exec_ackn(r)
   END DO
END SUBROUTINE creation_thread
```

Figure 5.8: Compilation Template for the Creation Thread

all initialization task have been finished (`ors_wait_creation_finished`). After having received the signal, the new SDA context has been established and the creation thread sends back an execution acknowledgment. From now on the new SDA object is accessible via the SDA variable the create method has been applied to (step 5.). The creation thread terminates on receipt of a termination request issued by the finalization routines of Opus.

5.2.3.2 The Execution Thread

The execution thread (cf. Figure 5.9) is realized as a Fortran 90 subroutine that contains pointers to the shared memory area (the MI-queues), the internal (distributed) data of an SDA, all of the method code that is unaltered by the Opus compiler except for the method calls (see below), as well as the code required to evaluate the condition clause for conditional method invocation. The execution thread also contains a generic "execution loop" that implements the state transitions of the SDA thread.

Specifically, the execution loop is the subroutine body of the execution thread and all method procedures are compiled to internal subroutines. Similarly, the condition clauses are transformed to internal logical functions. Hence, all method-subroutines and condition clause-functions have access to the internal SDA data as well as the MI-queues via Fortran 90 host association.

Once the execution thread is created by the creation thread all the internal SDA data is allocated through the Fortran 90 *automatic data object* mechanism and (if necessary) distributed by the underlying HPF system. Subsequently, the ORS is asked to establish the MI-queues as well as the environment for the underlying HPF system. If specified, the optional constructor method is executed.

```
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)
   <CALL constructor>
   CALL ors_signal_creation_finished
   execution_loop: DO
      exec_rec = ors_get_rec(queue=inqueue)

      SELECT CASE(exec_rec%method_id)
      CASE (1)
         IF (method_1_condition()) THEN
            CALL method_1(exec_rec%arguments ...)
            CALL ors_send_exec_ack(exec_rec)
            ! send results
            CALL ors_delete_record(exec_rec,inqueue)
         END IF
      ...
      CASE (n)
         IF (method_n_condition()) THEN
            CALL method_n(exec_rec%arguments ...)
            CALL ors_send_exec_ack(exec_rec)
            ! send results
            CALL ors_delete_record(exec_rec,inqueue)
         END IF
      CASE (termination_request)
         CALL ors_terminate_sda(outqueue)
         CALL ors_send_exec_ack(exec_rec)
         EXIT execution_loop
      END SELECT
   END DO execution_loop

CONTAINS
   ! method procedures and condition clause functions
END SUBROUTINE sda_name
```

Figure 5.9: Compilation Template for the Execution Thread

After these initialization steps the execution thread informs the creation thread that the SDA context has been established and enters the execution loop. Within the execution loop, the execution thread repeatedly checks the inqueue for MIs. If none are found, the execution thread is suspended (the *waiting* state of the SDA thread). Otherwise, the procedure associated with the MI is executed, given that the associated condition clause 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.

Specifically, the processing of an MI takes the following steps:

- If available, a record is retrieved from the inqueue (ORS procedure `ors_get_rec`).
- The condition clause function matching the indicated method (identified by the `method_id` component of the execution record), is evaluated (the *evaluating* state). If the condition-clause is satisfied, i.e., the function returns **true**, the MI is processed (the *executing* state) as follows:
  - The associated method is executed.
  - The ORS is asked to send back an execution acknowledgment.
  - The results are sent back to the caller.
  - Finally, the record is deleted from the inqueue.
- Otherwise (the condition-clause is not satisfied), another record, if available, is retrieved by calling `ors_get_rec` (the SDA thread re-enters the *waiting* state). Recall that the routine `ors_get_rec` is blocking, thus the execution thread is suspended if no ready requests are available and other threads (from the same or other contexts established on a node) may acquire access to the processor.

The execution thread terminates after having received a termination request. First, the ORS is asked to safely terminate the SDA (i.e. to synchronize with all outstanding requests) and then the execution loop (and thereby the execution thread) is exited.

5.2.3.3 The Server Thread

The primary task of the server thread, which is also created by the creation thread, is to receive and enqueue MIs while the method executions are carried out by the execution thread. However, 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 5.10 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. Depending on which method of which SDA returned, the appropriate results (i.e. results of function methods as well as parameters with intent **OUT** or **INOUT**) have to be received. Finally, the ORS is invoked in order to handle the return.

```
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)
         SELECT CASE (b%callee%type)
         CASE(1)
            SELECT CASE (b%method_id)
            CASE(1)
               ! receive results
            ...
            CASE(n)
               ! receive results
            END SELECT
         ...
         CASE(n)
            ...
         END SELECT
         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 5.10: Compilation Template for the Server Thread

- *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 and results 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.

5.2.3.4 Method Invocation

SDAs communicate with each other via method invocation requests and synchronize or exchange progress information by *waiting* or *testing* an event associated with an asynchronous method 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 5.11.

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. Similarly, distribution descriptors for all of the actual arguments are provided. This record is then inserted into the outqueue (recall that the outqueue is accessible via host association) 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 case of an asynchronous call, the caller resumes its execution immediately while in 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, more specifically, copied out of the local segment, 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 routine *wait* or *test* is simply compiled to a respective ORS call.

5.2.4 Interaction Between Distributed SDAs

To allow for data parallel execution, an SDA object may be distributed across multiple nodes. Thus, an SDA context contains a distributed SDA thread, that is,depending on the underlying system, a set of parameterized processes or lightweight threads. 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 [43, 59, 162] are employed for this task. However, in order to make use of these libraries, we have to ensure that:

1. the parameterized processes/threads on all the 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 is a problem of synchronizing all the threads/processes of an SDA context. 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 [101]) 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.

```
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>,<descriptors>)
  CALL ors_insert_rec(r,outqueue)
  CALL ors_send_synchronous_mi(r)
  ! send input arguments
  ...
! e = SPAWN b.m2(...)
  r = ors_new_rec(b,method_id(b%m2),<arguments>,<descriptors>)
  CALL ors_insert_rec(r,outqueue)
  CALL ors_new_event(e,r)
  CALL ors_send_asynchronous_mi(r)
  ! send input arguments
  ...
! wait(e)
  CALL ors_wait(e,ors_my_sda)
  ...
END SUBROUTINE method_1
```

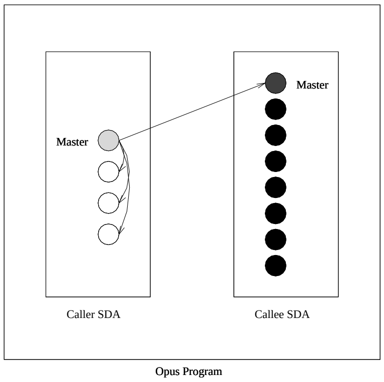
Figure 5.11: Compilation Template for Method Invocations

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[1].

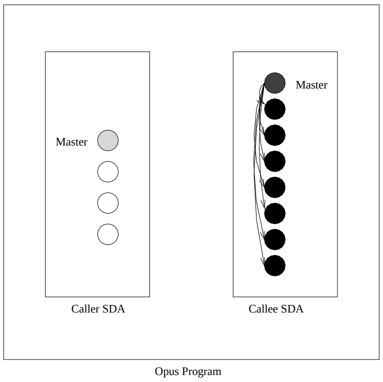
Having all nodes of the caller and callee agreed on performing the data transfer and having exchanged the necessary data layout descriptors, a redistribution library can be asked to compute communication schedules and to perform the actual data transfer. Figure 5.12 illustrates the data exchange process. Note that step one to three on the callee side takes place in the server threads; hence communication may well be overlapped by computation taking place in the execution threads.

Another synchronization issue is that all execution threads of an SDA need to work on the same request. 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 condition clauses 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.

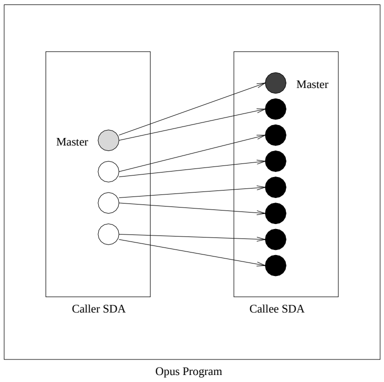
[1] Dynamic dummy arguments as defined in the Approved Extensions of HPF-2 would require a transmission of the actual layout descriptors from the callee to the caller since they might have changed. However, this extension is currently not supported in Opus.



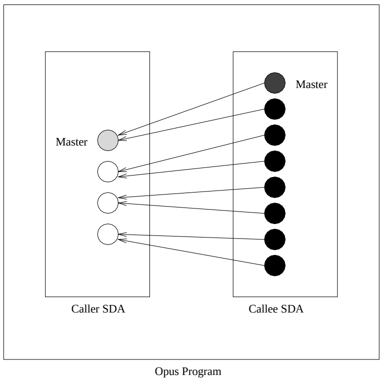
1. Caller master sends an MI to callee master with actual argument distributions and notifies its workers.



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

Figure 5.12: Illustration of the MI Process for Distributed SDAs

5.2.5 Practical Remarks on the Implementation

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

5.2.5.1 Thread Compliance

Our design, as described above, assumes a multithreaded system. This requires the underlying message passing system (which we assume to be MPI without loss of generality) to be thread compliant as specified in the MPI-2 standard [141]. However, current public domain MPI implementations, such as *mpich* [94], 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 when employing a non-thread-safe MPI implementation. 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 also carries out the responsibilities of the creation thread. One of the issues that arise is that the server thread cannot block on a call to the `ors_receive_mi` routine, since it is servicing calls for multiple SDAs and has to handle all outgoing messages as well. Hence, the server thread has to actively poll for messages which may cause a significant overhead. Fortunately, many vendors such as SUN, IBM, and SCALI offer thread-safe MPI implementations.

In addition, the underlying HPF runtime and communication systems are also required to be thread-safe. If their thread-safety cannot be guaranteed an appropriate mutex protection could be considered provided that no blocking routines that might block the whole system are involved.

5.2.5.2 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 requests 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 actively polls for requests, instead.

Another problem within the context of process based implementations is that the underlying message passing system must be able to support dynamic process management as specified in the MPI-2 standard. Unfortunately, this is, for example, not possible with the current implementation of *mpich* [94]. An implementation based upon MPI-2 compliant systems, such as LAM-MPI [124], could be considered instead.

5.2.5.3 Heterogeneous Environments

Even though our current implementation executes Opus programs in a homogeneous environment—more specifically, in a *virtually* homogeneous environment—there is nothing inherently in the design which prohibits execution in a heterogeneous environment. Note that with the term *heterogeneous environment* we refer to complex clusters which do not support a common communication layer. In particular, this view of heterogeneity does not cover clusters of heterogeneous workstations since for instance the *mpich* or *LAM* implementation of MPI already provide a *parallel virtual machine* view of the cluster. As a consequence, 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 system support for interconnecting machines efficiently is required (see also [133]). These systems include message passing systems for heterogeneous platforms, such as IMPI [115, 168], PACX-MPI [17], PVMPI [68], and Nexus [82]. Java's networking facilities may also be exploited. In particular, our OpusJava framework (cf. Chapter 4) may not only interconnect HPF and Java SDAs, but also pure HPF SDAs deployed in a heterogeneous environment. In such a scenario, requests between SDAs located on different components of a heterogeneous system are routed through the External SDA Stubs of

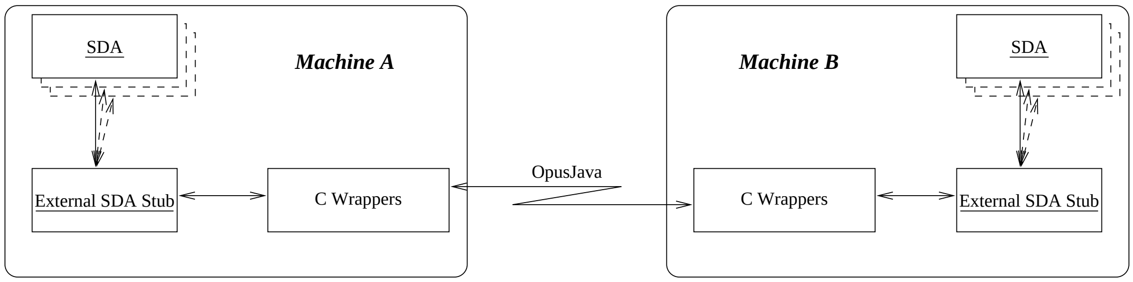


Figure 5.13: OpusJava as Transportation Layer in Heterogeneous Environments

both, the caller and the callee, translated to Java, and transmitted by the OpusJava mechanism (cf. Section 5.3). Figure 5.13 illustrates such a scenario. As already discussed in Chapter 4, this scenario involves expensive data marshaling overheads due to Java's array concept.

5.2.5.4 HPF Compilation

The output of the Opus compiler is an HPF program augmented with calls to the ORS. This HPF program needs to be compiled by an HPF compiler in order to produce an executable program. We have designed our implementation to work in close cooperation with a specific HPF compiler—VFC [21]. However, there is nothing inherently in our design which prohibits the integration of HPF compilers others than VFC. In particular, we designed SDAs as wrappers around HPF programs which could be compiled with any standard HPF compiler. Our design of the Opus system even allows different SDAs to be compiled by different HPF compilers since the Opus coordination layer is independent of the underlying HPF program. However, on the runtime level interoperability issues arise. First, we require some means to inform the HPF runtime system about the resources available to an SDA. If no direct access to the HPF runtime system is possible, the HPF-2 tasking mechanism could be exploited for this purpose. Second, our implementation of inter-SDA communication exploits the distribution information available in the HPF runtime system. Again, the HPF runtime system should be able to expose this information to the Opus runtime system. Alternatively, mapping inquiry subroutines as defined for HPF-2 may be employed. Finally, the employed HPF runtime system has to be thread-safe, as already discussed above.

5.2.6 Implementation Rationale

In this section, we highlight 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.

5.2.6.1 System Support

In Section 5.2.2, we identified the functionalities 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 [99], Nexus [83], or Panda [30]. 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 [98]. 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 [77], 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 (for example, MPI, TCP/IP) and multithreading (for instance, 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.

5.2.6.2 The Opus Runtime System and the Opus Compiler

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 [100], it was 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 assumed a fairly sophisticated runtime system based upon Chant [99]. However, it had some major drawbacks, in particular the interoperability problems among the different runtime components employed, as discussed above. Moreover, invoking HPF routines from within C/C++ is a major problem since compiler dependent information (in particular data distribution descriptors) have to be passed to an HPF routine. In our current approach the whole call site is transformed to HPF code that is subsequently processed by an HPF compiler.

The same problems arise when using other high level coordination libraries like KeLP [72] which is also a C++ library that needs additional conversion of the HPF data and HPF mapping descriptors into its own convention. Again, interoperability problems with the VFC runtime system would occur, as described in the previous subsection.

The HPF binding of MPI (HPF/MPI [84]) 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. Moreover, the compiler approach allows an easier integration of new language features, such as those described in Chapter 3, whose implementation and integration in the Opus system is discussed below.

Related coordination approaches that make use of a compiler are, for example, Fx [96], Fortran-M [75], or Orca [11]. 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 [18] 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.

5.2.7 Advanced Opus Features

After having described the implementation design of the basic Coordination Language Opus above, we now survey issues that arise in the implementation of the Advanced Opus Features discussed in Chapter 3. We present both, compilation techniques and the required runtime support. Moreover, deviations from the general design presented above will be discussed.

5.2.7.1 Intra-SDA Parallelism

As described in Section 3.2, intra-SDA parallelism can be described by means of *ParBlock* directives. The Opus compiler parses the ParBlock directives of an SDA type and generates a ParSet for each method x of an SDA. The ParSet indicates with which other methods x is allowed to be executed concurrently.

Compilation

Algorithm 5.2.1 describes the generation of ParSets: first, the ParSet of method x is initialized to an empty set. Subsequently, all the par-sections of all the ParBlocks of the SDA type are inspected and, if x is part of the examined par-section, the contents of the par-section is added to the ParSet. Note that x itself is only added, if it appears more than once within a par-section. Finally, if x is a pure method, all other pure methods (including x itself) added to the ParSet, as well. ParSets can be efficiently implemented by means of bit-vectors.

Algorithm 5.2.1: Computation of ParSets

Notation: Let $\mathcal{M}$ denote the Set of all methods of an SDA type. $\mathcal{PB}$ denotes the set of all ParBlocks defined for an SDA type and $PS \in \mathcal{PB}$ a par-section. The ParSet of a method $x \in \mathcal{M}$ is denoted by $\mathcal{P}^x$. $\emptyset$ is the empty set.

do $\forall\ x\ \in\ \mathcal{M}$
  /* initialization of $\mathcal{P}^x$ */
  $\mathcal{P}^x\ =\ \emptyset$;

  /* loop over all par-sections */
  do $\forall\ PS\ \in\ \mathcal{PB}$
    if $(x\ \in\ PS)$
      if (appears_multiple_times(x, PS))
        $\mathcal{P}^x\ =\ \mathcal{P}^x\ \cup\ PS$
      else
        $\mathcal{P}^x\ =\ \mathcal{P}^x\ \cup\ (PS \setminus \{x\})$
      end if
    end if
  end do

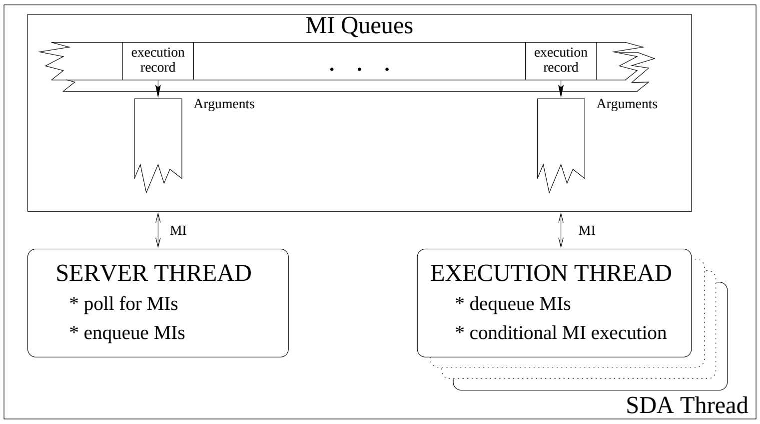


Figure 5.14: Structure of an SDA Thread with Multiple Execution Threads

```
    if (is_pure(x))
      /* loop over all methods */
      do ∀ m ∈ M
        if (is_pure(m))
          P^x = P^x ∪ {m}
        end if
      end do
    end if
  end do
```

■

Runtime Support

All control activity related to an SDA is executed within the *SDA thread*. As discussed in Section 5.2.3, a single SDA thread ensures the default monitor semantics of SDAs. However, to allow concurrent method executions within an SDA, an SDA context requires a set of SDA threads, an SDA thread pool. Recall that an SDA thread is comprised of two independent lightweight threads (the server and execution thread) as well as a shared memory segment for storing MI-queues. An SDA thread pool does not require multiple instances of all these components but only multiple execution threads. Therefore, an SDA thread pool is in essence realized by an *execution thread* pool as depicted in Figure 5.14.

The total number of execution threads is system dependent: while on an SMP the number of execution threads should at least be equal to the number of available processors, on a single processor machine only a few of them are needed. The actual number can be determined dynamically at runtime via environment variables.

The execution threads are in duty of validating whether a new method can start executing in parallel with others. For this purpose we keep track of all methods currently being executed. A new method can start its execution if and only if the set of the currently executing methods is a subset of its ParSet. In addition, its condition clause must be satisfied as well. Obviously, both checks need to be performed in an atomic action in order to avoid inconsistencies due to overlapping checks. Therefore, the compilation template of the execution thread (cf. Figure 5.9) has to be slightly modified. Algorithm 5.2.2 illustrates the modified method invocation mechanism of an execution thread.

Algorithm 5.2.2: Modified Execution Thread

Notation: Let Π denote the set of all methods currently being executed.

...
/* begin atomic */
if ($\Pi \subseteq \mathcal{P}^x$)
  /* Concurrency control allows the execution */

  if (/* condition-clause(x) */)
    /* Condition clause is satisfied as well */

    $\Pi = \Pi \cup \{x\}$
/* end atomic */

    /* launch execution of x */

    /* execution of x has finished */
    $\Pi = \Pi \setminus \{x\}$
  end if
end if
...

■

5.2.7.2 Lightweight SDAs

Lightweight SDAs introduced in Section 3.3 provide a means for *passive* objects that may only exist within the context of a standard SDA with which they share resources. The key property of lightweight SDAs is their reduced creation and method invocation overhead.

Compilation

The compilation of lightweight SDAs is quite different to that of standard SDAs. While standard SDAs are compiled to *active objects* comprised of two separate lightweight threads (the server and the execution thread) according to pre-defined compilation templates, lightweight SDAs are compiled to *passive objects* by exploiting techniques that simulate object oriented features in Fortran 90 [64].

All lightweight SDAs declared within an Opus program are packed into a Fortran 90 module whose data is the internal data as well as the runtime representation of the lightweight SDAs and whose procedures are the methods of the lightweight SDAs. Specifically, the internal data of lightweight SDAs is packed into a derived type to allow multiple instantiations of the lightweight SDA. Of course, this modification of the SDA data requires also modifications of the methods, in particular, the internal data of the SDA is no longer available via host association but must be passed as arguments to the methods.

Example 5.2.1: Compilation

The following lightweight SDA type:

```
SDA TYPE, LIGHTWEIGHT :: foo
   INTEGER bar
CONTAINS
   SUBROUTINE sub(x)
      INTEGER, INTENT(IN) :: x

      bar = x
   END SUBROUTINE sub
END SDA TYPE foo
```

is translated to the following entry in the lightweight sda module:

```
MODULE lightweight_sdas
   TYPE foo_data
      INTEGER bar
   END TYPE foo_data

   TYPE sda_foo
      ! management data
      TYPE (foo_data), POINTER :: data
   END TYPE sda_foo
CONTAINS
   SUBROUTINE foo_sub(d,x)
      TYPE(foo_data) d
      INTEGER, INTENT(IN) :: x

      d%bar = x
   END SUBROUTINE foo_sub
END MODULE lightweight_sdas
```

■

The lightweight SDAs module which contains all information of all the lightweight SDA types in an Opus program is use associated with any standard SDA in the Opus program.

Runtime Support

In the following we briefly sketch the runtime behavior of lightweight SDAs, in particular their creation and method invocation mechanisms.

Lightweight SDA Creation. The creation of a lightweight SDA involves only the allocation of the internal data of the SDA as well as of some management data. This management data in particular includes information about the base SDA such that remote SDAs having access to the lightweight SDA may invoke its methods by contacting the sever thread of the base SDA the lightweight SDA is associated with.

Example 5.2.2: Lightweight SDA Creation

Recall the SDA definition from Example 5.2.1. The following code excerpt illustrates how the creation of an SDA `foo` *from within a standard SDA is accomplished. Note that we only sketch the execution thread of the standard SDA.*

```
SDA TYPE standard
   SDA (foo) :: l
   ...
CONTAINS
   SUBROUTINE init
      CALL l.create()
      ...
   END SUBROUTINE init
   ...
END SDA TYPE standard
```

From the SDA type definition above, the following execution thread is derived:

```
SUBROUTINE standard
   USE lightweight_sdas

   TYPE(sda_foo) :: l
   ...
CONTAINS
   SUBROUTINE init
      CALL ors_init_lightweight_sda(l)
      CALL ors_allocate_foo_data(l)
      ...
   END SUBROUTINE init
   ...
END SUBROUTINE standard
```

The runtime routine `ors_init_lightweight_sda` *(a generic Fortran interface which is generated by the compiler) initializes the necessary management data and informs the server thread that it may from now on serve remote requests for this SDA. Subsequently, all internal data of the lightweight SDA is allocated by a call to the compiler generated routine* `ors_allocate_foo_data`*.*

■

Method Invocation. In discussing the method invocation mechanisms for lightweight SDAs, we have to distinguish between two issues: first, the method invocation request may be a *synchronous* or *asynchronous* one, and second, the request may be *local* or *remote*. We first discuss synchronous requests and focus on the locality issue.

Locality in Method Invocations: A method invocation request for a lightweight SDA is said to be *local* if (1) it originates from an SDA that belongs to the same SDA Tree as the callee and (2) the nodes allotted to the caller are a superset of the nodes of the callee. Otherwise the request is said to be *remote*.

The reason for this differentiation is that arguments of local calls are available on all the nodes allotted to the lightweight SDA and hence standard Fortran/HPF procedure calls may be used. Of course, depending on the data distribution specified with HPF, redistribution on procedure entry and exit might be necessary. In contrast, it is not possible to implement remote invocations as simple procedure calls since caller and callee act on disjoint memory segments and disjoint sets of nodes may be associated with them. Consequently, calls have to be routed through the server thread of the callee's base SDA.

Example 5.2.3: Synchronous Invocation

In the code excerpt below, which is a continuation of Example 5.2.2, we find two synchronous calls. The first one affecting SDA `l` *is known to be a local one, since* `l` *is a child SDA of the caller. SDA* `ll`*, however, is passed as an argument to subroutine* `work` *hence its locality cannot be statically determined.*

```
SDA TYPE standard
   SDA (foo) :: l
   ...
CONTAINS
   ...
   SUBROUTINE work(ll)
      SDA (foo) :: ll

      CALL l.sub(1)
      CALL ll.sub(2)
      ...
   END SUBROUTINE work
END SDA TYPE standard
```

The generated execution thread reflects the differences between `l` *and* `ll`*. Calls to local SDAs are compiled to simple procedure calls whereas calls to remote SDAs are implemented in the standard manner.*

```
SUBROUTINE standard
   USE lightweight_sdas

   TYPE(sda_foo) :: l
   TYPE(ors_execution_rec), POINTER :: r
   ...
CONTAINS
   ...
   SUBROUTINE work(ll)
      TYPE(sda_foo) :: ll

!! call l.sub(1) !!
      CALL foo_sub(l%data,1)
!!!!!!!!!!!!!!!!!!

!! call ll.sub(2) !!
      IF (is_local(ll)) THEN
         CALL foo_sub(ll%data,2)
      ELSE
         r = ors_new_rec(ll,...)
         CALL ors_insert_rec(r,outqueue)
         CALL ors_send_synchronous_mi(r)
      END IF
!!!!!!!!!!!!!!!!!!!
      ...
   END SUBROUTINE work
END SUBROUTINE standard
```

■

While synchronous method invocations can be implemented quite straight-forward as illustrated above, asynchronous ones need some special handling. The main problems with respect to asynchronous requests arise in the context of local invocations. Remote asynchronous invocations have to be implemented with the standard Opus mechanism for the same reasons as for remote synchronous invocations.

Since all SDAs of an SDA tree share the same resources, in particular the SDA threads, the degree of parallelism is restricted to the number of SDA threads allotted to the base SDA. If only one SDA thread is available the request will probably be sequentialized and it can be implemented as a simple procedure call just as if it were a sequential request. However, some parallelism might still be exploitable in that the lightweight SDA might not run on all the nodes of the base SDA. Hence, the tasking features of HPF-2 [107] can be exploited to invoke the method only on the processors allotted to the lightweight SDA.

In the case of multiple SDA threads (or if new SDA threads are dynamically spawned upon request), another thread than the one executing the call may eventually execute the method. Therefore, a standard method invocation request is generated which is inserted into the local inqueue from where it can be retrieved by any idle SDA thread. Note that although an execution record is generated the input data is not copied but only pointers to it are passed to the record.

Example 5.2.4: Asynchronous Invocation

The following code excerpt illustrates asynchronous method invocations in local and (potentially) remote SDAs:

```
      SDA TYPE standard
!HPF$ PROCESSORS P(number_of_processors())
         SDA (foo) :: l
         ...
      CONTAINS
         ...
         SUBROUTINE work(ll)
            SDA (foo) :: ll

            SPAWN l.sub(1)
            SPAWN ll.sub(2)
            ...
         END SUBROUTINE work
      END SDA TYPE standard
```

As described above, the invocation in `l` *is a local one, hence we have to distinguish between a multiple SDA thread case and a single SDA thread case. While for the former one an execution record is created and inserted into the local in- and outqueues, the latter case is implemented by means of an HPF* `TASK_REGION`*.*

The second call may be a local or a remote one. In the local case we have again to distinguish between a multiple SDA thread case and a single SDA thread case. Both situations are implemented in the same way as described above. The remote case is implemented with the standard Opus mechanism. Note that arguments are only copied in case of a remote call.

```
      SUBROUTINE standard
         USE lightweight_sdas
!HPF$ PROCESSORS P(number_of_processors())
         TYPE(sda_foo) :: l
         TYPE(ors_execution_rec), POINTER :: r
         ...
      CONTAINS
         SUBROUTINE work(ll)
            TYPE(sda_foo) :: ll

      !! spawn l.sub(1) !!
            IF (multiple_threads) THEN
               r = ors_new_rec(ll,...)
               CALL ors_insert_rec(r,outqueue)
               CALL ors_insert_rec(r,inqueue)
            ELSE
!HPF$          TASK_REGION
!HPF$             ON P(l%processors) CALL foo_sub(l%data,1)
!HPF$          END TASK_REGION
            END IF
      !!!!!!!!!!!!!!!!!!!!

      !! spawn ll.sub(2) !!
            IF (is_local(ll)) THEN
               IF (multiple_threads) THEN
                  r = ors_new_rec(ll,...)
                  CALL ors_insert_rec(r,outqueue)
                  CALL ors_insert_rec(r,inqueue)
               ELSE
!HPF$             TASK_REGION
!HPF$                ON P(ll%processors) CALL foo_sub(ll%data,2)
!HPF$             END TASK_REGION
               END IF
            ELSE
               r = ors_new_rec(ll,...)
               CALL ors_insert_rec(r,outqueue)
               CALL ors_send_asynchronous_mi(r)
            END IF
      !!!!!!!!!!!!!!!!!!!!!
            ...
         END SUBROUTINE work
      END SUBROUTINE standard
```

■

Condition Clauses. So far we have only considered lightweight SDA methods that are not protected by a condition clause. The presence of condition clauses requires some special care in case of local invocations that are handled by the calling thread. As can be seen from the discussion above, such method invocations can be implemented by means of a standard procedure call if no condition clause guards the method's execution. Otherwise, the implementation needs to take care that after an unsuccessful test of the condition clause the calling SDA thread is not blocked in waiting for the condition to occur but is able to execute any other request to be handled by the same SDA thread. This functionality can be provided by implementing any sort of task scheduler. If sufficient SDA threads are available (or if new SDA threads are dynamically spawned if required), the calling thread may well block.

5.2.7.3 SDA Migration

Migrating an SDA (cf. Section 3.4) implies the re-location of all the SDA's internal data as well as its runtime information. More specifically, an exact copy of the SDA, including its MI-queues, needs to be created on the new resources.

The implementation of migration involves five steps. Suppose an SDA S of type A which was created on resources *R1* should migrate to resources *R2*. The intrinsic method `migrate` is compiled to the following sequence of steps:

1. Set up a new SDA object S' of type A on *R2*.
2. Copy the runtime data (inqueue, outqueue) from S to S'. This step requires advanced runtime support since it involves serialization and communication of dynamic queues.
3. Copy all internal data of S to S'. This step is implemented similar to data exchange among distributed SDAs (cf. Section 5.2.4).
4. Deallocate the internal data of S and terminate its execution thread.
5. Inform the server thread of the new location of the SDA.

As already mentioned in Section 3.4, all these steps require exclusive access to the SDA and are performed as an atomic action. Hence, requests that arrive while the migration is in course will already be forwarded to the new location of the SDA.

Note that SDA migration is different to *thread-migration* [61] where active threads are relocated. Instead, threads of an SDA are terminated and restarted. Thread migration in the context of data parallel programs would require a dynamic update of the underlying message passing system (such as MPI), which is in most cases not possible.

After the migration has taken place, a *stub* of the migrated SDA remains on the original resources. This stub is in duty of routing requests that might still arrive at the old resources to the new location of the SDA. More specifically, the *server thread* of the SDA remains active on the old resources but instead of servicing requests it forwards the request to the new resources. This functionality of the server thread requires runtime checks whether an SDA has migrated or not. Since all SDAs that might migrate must be annotated with the **DYNAMIC** attribute, this check can be relaxed in all SDAs not having this attribute.

Instead of *forwarding* requests, *bouncing* of requests could also be considered. In this case a request together with the new location of the migrated SDA would be returned to its originator which in turn could re-submit the request. Request bouncing requires more communication than forwarding, however, it is advantageous if misrouted requests from an SDA are issued concurrently. Once a bounced message is received the SDA can update its information about its callee and all subsequent messages can be sent directly to the new location, while in case of forwarding, the update can only take place after the results of the computation have been received from the new location.

Figure 5.15 illustrates request forwarding (A) and request bouncing (B). In the first case a request sent from SDA `A` to SDA `S` (a) is forwarded by the stub to the new SDA `S'` (b). Once the results for this request are received (c) SDA `A` updates its information and subsequent requests are sent to `S'` directly (d). In the latter case the request is bounced (b) such that it has to be re-submitted to `S'` (c). Again, results are transmitted directly (d) and subsequent requests use the updated location information (e).

5.2.7.4 Priority Scheduling

The implementation of priority scheduling (cf. Section 3.5) requires a modification of the strategy of how the execution thread retrieves records from its inqueue. The basic implementation design

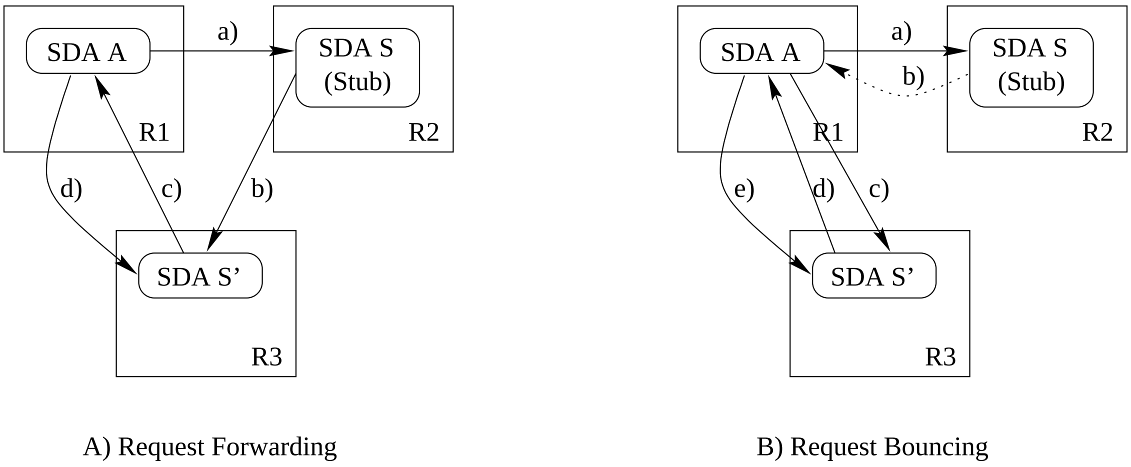


Figure 5.15: SDA Migration: Request Routing

assumes that the inqueue is realized as a FIFO queue. In consequence, requests are served relative to their arrival order (given satisfied condition clauses). Under priority scheduling, the request have to be served relative to their priority value, instead. This is best accomplished by implementing the inqueue using hashing strategies, where the hashing is based upon the priority value, instead of FIFO strategies. Apart from this change in the inqueue no other modifications to our basic design are required. In fact, even with a hash based implementation FIFO strategies may still be applied for requests having the same priority value.

5.3 Implementing OpusJava

In this section we provide an overview of the implementation design of OpusJava (see also [129]). As described in Chapter 4, the OpusJava framework comprises two main components: a Java class library that facilitates Java SDAs and its interface to HPF. While the former one is designed as a pure Java package, the latter one consists of a Java part, JNI interfaces, and Fortran components. All the parts of the HPF interface are generated automatically by the Opus compiler.

In implementing OpusJava we followed closely the implementation design of the Coordination Language Opus which was described above. Consequently, we make use of Java's multithreading features for implementing SDA objects. Communication among SDAs is facilitated by Java's Remote Method Invocation (RMI) mechanism as well as by socket communication. Java's object serialization mechanisms are employed for data marshaling. It has often been argued that Java's RMI and object serialization mechanisms imply considerable overheads. Although this objection is certainly true for the time being, Java implementations are rapidly being improved so that we can quite safely assume that these overheads will be reduced. First attempts in this directions have been reported in [136, 146, 159].

In the following, we discuss the design of the OpusJava class library as well as of its interface to the Coordination Language Opus in detail.

5.3.1 The OpusJava Class Library

The OpusJava class library facilitates the creation of any Java class as an SDA. It implements the classes `SDA`, `Event`, `Resource`, and `Request` which can be used by the end-user. Their API can be found in Appendix C. Apart from these classes, some supporting mechanisms are required. More specifically, the OpusJava class library comprises three main components:

```
public interface SDAImplInterface extends Remote {

    public Object insertSync (String name, Object args[])
        throws RemoteException;

    public int insertAsync(String name, Object args[], int caller_port)
        throws RemoteException;

    public String migrate(SDAServer remoteServer)
        throws RemoteException;

    public void save(String filename, String codebase)
        throws RemoteException;
}
```

Figure 5.16: Interface of the SDA Implementation Skeleton

- SDA Implementation Skeletons, which are generic skeletons for SDA objects,
- SDA Stubs, which are pointers to SDA objects, and
- SDA Servers, which are daemons, responsible for creating and terminating SDA objects.

The SDA Implementation Skeleton

Every Java SDA object is wrapped by a skeleton that handles all its interactions with other components. In particular, methods of an SDA are invoked via the skeleton rather than directly in the SDA. Skeletons are implemented by the final Java class `SDAImpl`. Figure 5.16 shows the interface of SDA implementation skeletons. In order to allow remote access to SDAs, skeletons are remote objects that are registered in the RMIRegistry. Four methods are exported: `insertSync`, `insertAsync`, `migrate`, and `save`. `insertSync` and `insertAsync` are responsible for receiving and temporarily storing synchronous and asynchronous method invocation requests, respectively. Note that a port is passed to `insertAsync`. This port is used to transfer results of the method execution back to the caller. Method `migrate` implements SDA migration to a new Server and returns the name under which the migrated SDA is bound in the new RMIRegistry. Persistence of an SDA is implemented in method `save` which dumps an SDA to a file in a given codebase exploiting Java's object serialization mechanism.

`SDAImpl` skeletons are realized in a manner similar to Opus SDA objects (cf. Section 5.2). In particular, a skeleton contains an *inqueue* for managing *execution records* and operates an *execution thread* which is spawned upon creation. Additionally, information about the progress of asynchronous requests is stored in a hashtable called *eventtable*, such that objects having access to the `Event` associated with the asynchronous call may inquire the progress of the request (see the discussion of Events below). Method invocation requests are processed in a similar way as in the Opus *Server Thread*: the `insert` methods create an *execution record* which contains references to the method to be executed, the input parameters, and some internal management data. Java's reflection mechanism is exploited for the creation of execution records. In particular, the correctness of the supplied method signature is validated. The execution record is then inserted into the *inqueue* from where it is retrieved by the *execution thread*. After ensuring the validity of the request by evaluating the associated *condition clause* the execution thread invokes the method in the SDA object. The skeleton takes also care of transferring results of a method execution back to the caller. In case of a synchronous call the results are embodied in a Java object returned as the result of `insertSync`. `insertAsync` does only return the ID of the eventtable entry immediately; the results of the method execution are communicated separately using plain socket communication when the method execution eventually took place.

The SDA Stub

Any Java object may access an SDA object via a local SDA stub that is realized by the final class `SDA`. This class is accessible by the user and was introduced in Chapter 4. An SDA stub is in fact a pointer to an `SDAImpl` object. SDA stubs take care of transmitting requests as well as receiving results synchronously or asynchronously. The constructor of a stub issues an RMI call to the *SDA Server* (cf. next paragraph) on the indicated machine (available via `Resource` objects) and thus initiates the construction of a new SDA object together with an associated `SDAImpl` skeleton which is registered in the RMIRegistry. A permanent RMI connection for subsequent method invocations is built between the `SDA` stub and the `SDAImpl` skeleton.

An SDA stub defines six methods: `call`, `spawn`, `equals`, `terminate`, `migrate`, and `save`. Their API can be found in Appendix C. Activations of the `call` method results in an RMI request to the skeleton of the callee. The results of this RMI request are the results of the method execution, embodied within a Java object. The caller is blocked until the results are available. In contrast, an activation of the `spawn` method first creates a new `Event` object which opens a socket for retrieving the results and spawns a new thread that listens on this socket. Again, an RMI call to the skeleton is made. This call returns immediately without waiting for the execution of the method. Thus, the caller is able to resume its execution and to run asynchronously with the callee. It is however necessary to wait until all input arguments have been transferred in order to avoid race conditions. Hence, the `spawn` method only returns after the RMI call to the skeleton returns thus ensuring that all arguments have been copied out of the local segment.

The `equals` method evaluates whether the local `SDAImpl` reference and the reference of the `SDA` object passed as argument refer to the same `SDAImpl` object.

An activation of the `terminate` method allows SDAs to terminate gracefully. In particular, the termination of an SDA requires its synchronization with all outstanding method invocations. Subsequently, the `terminate` method of the SDA Server is activated which results in the deletion of the SDA from the RMIRegistry.

The `migrate` method activates the corresponding `migrate` method in the SDA skeleton and receives the name under which the migrated SDA is bound in the new RMIRegistry as result. With help of this new name the RMI connection can be updated.

An activation of the `save` method simply executes the `save` method in the skeleton.

The SDA Server

The SDA Server is a daemon that has to be installed on any system participating in an OpusJava program execution. This daemon is responsible for creating and terminating SDA objects for which four RMI methods (`create`, `createMigrated`, `load` and `terminate`) are exported. Figure 5.17 depicts the interface of an SDA server.

The `create` method of the SDA Server, which is activated by the constructor of an `SDA` stub, has 2 arguments: the name of the SDA type and an array of input parameters for the SDA's constructor. Any Java object can be created as an SDA and Java's reflection mechanism is exploited to check whether appropriate classes are available. If not, dynamic class loading can be used to load the classes from user specified resources on demand. Subsequently, the new SDA object is created using the supplied arguments for its constructor. An associated skeleton is created and registered in the RMIRegistry, as well. The registered name is sent back to the caller, thus completing the `create` method.

Other possibilities for establishing a new SDA object are the `load` and the `createMigrated` methods. `load` restores a previously stored object from persistent memory and is, similar to `create`, activated by the constructor of an `SDA` stub. Method `createMigrated` is activated by the `migrate` method of an `SDAImpl` skeleton and restores a migrated SDA. It has three input arguments, the object to be restored, the inqueue and the eventtable operated by the skeleton.

```
public interface SDAServerInterface extends Remote {

    public String create(String name, Object args[])
        throws RemoteException;

    public String createMigrated(Object o,ArrayList in, Hashtable events)
        throws RemoteException;

    public String load(String filename, String codebase)
        throws RemoteException;

    public void terminate(String name)
        throws RemoteException;
}
```

Figure 5.17: Interface of the SDA Server

The `terminate` method simply deletes an SDA skeleton from the RMIRegistry as sketched above. Thus, Java's garbage collector may claim the associated memory when necessary.

Example 5.3.1: SDA Creation and Method Invocation

Consider the following simple OpusJava program skeleton which creates a new SDA of type `UserSDA` *on Machine B and synchronously invokes a method.*

```
...
Resources r = new Resources("B",1)
SDA foo = new SDA(UserSDA,...,r)

Object result = foo.call(...)
...
```

Figure 5.18 summarizes and illustrates the mechanisms involved in the creation of the SDAs and in the synchronous method invocation:

1. *A new* `SDA` *object is created on the caller's side.*
2. *The SDA Server of the target machine is contacted via RMI resulting in*
3. *the creation of the SDA object (*`UserSDA`*), as well as*
4. *an associated SDAImpl object, which contains a reference to the SDA object; the execution thread is spawned.*
5. *The SDAImpl object is registered in the RMIRegistry and*
6. *the registered name is sent back to the caller.*
7. *For invoking a method synchronously, the SDA stub issues an RMI call to the* `insertSync` *method of the associated SDAImpl skeleton.*
8. *The* execution thread *of the skeleton executes the method and*
9. *sends back the results, thus completing the method invocation.*

■

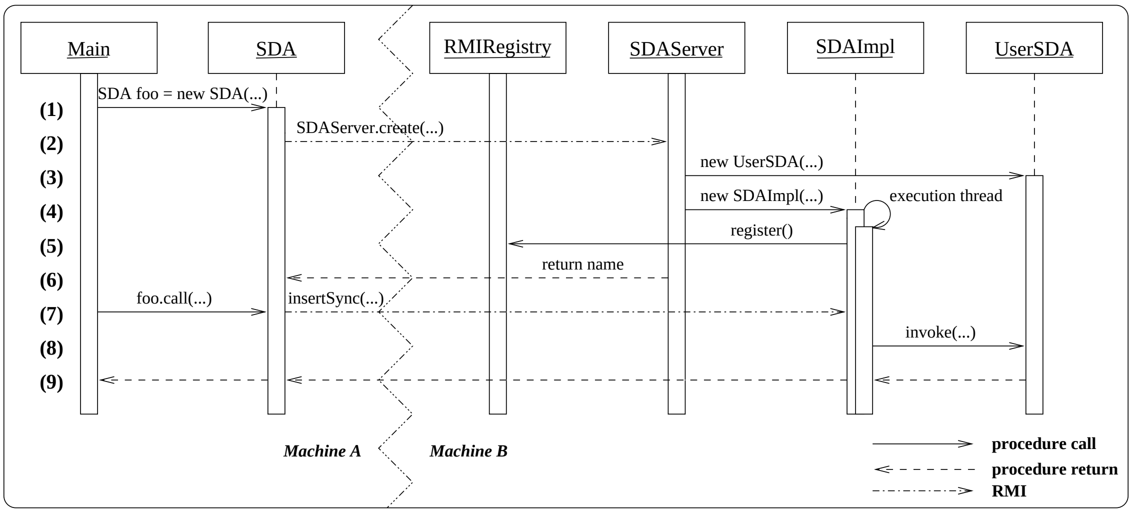


Figure 5.18: UML Sequence Diagram for SDA Creation and Synchronous Method Invocation

Events and Resources

Apart from the main components described above, the OpusJava framework implements `Event` and `Resource` objects. While `Resource` objects are simple containers for IP-addresses, `Event` objects are conceptually pointers to asynchronous method invocations. `Event` objects are created in course of asynchronous method invocations. An event contains a `finished`-flag that indicates the progress of the associated request as well as a `result` field which is used to store the output parameters of the associated method. The `result` field is written by a separate thread that listens on a specific socket waiting for the results. Events have three public methods, `getResults`, `testEvent`, and `waitEvent`. Their semantics have already been described in Chapter 4. Although `Event` objects are created by the SDA stub executing the call, the SDA skeleton receiving the request stores information about its progress in a hashtable called *eventtable*. The key used in this hashtable is stored in the associated `Event` object such that objects other than the calling stub which hold a copy of the `Event` may contact the skeleton serving the request in order to retrieve the request's progress information for synchronization purposes. This procedure is necessary since the `finished` flag of `Event` copies cannot be updated automatically.

Example 5.3.2: Asynchronous Method Invocation

In continuation of Example 5.3.1 the following code excerpt illustrates an asynchronous method invocation and the associated Event handling:

```
...
Event e = foo.spawn(...)
Object result = e.getResults()
...
```

Figure 5.19 illustrates the asynchronous method invocation together with the involved Event mechanism:

1. *The activation of the* `spawn` *method causes the creation of a new* `Event` *object that opens a new socket and spawns a thread that listens on this socket.*

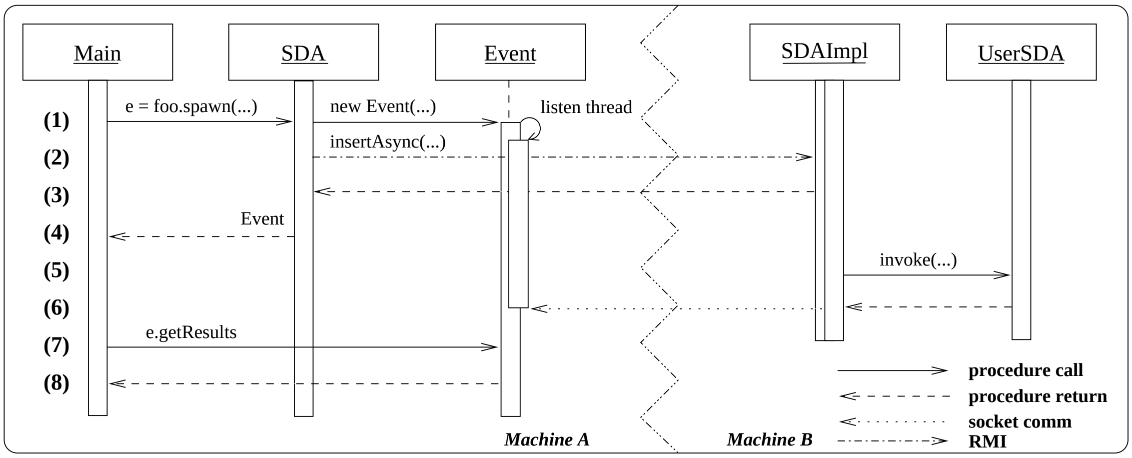


Figure 5.19: UML Sequence Diagram for Asynchronous Method Invocation and Event Handling

2. *Subsequently, an RMI request is issued. This request results in the insertion of an* execution record *and an* eventtable entry *in the skeleton, after which*
3. *the RMI call returns passing the key of the eventtable entry.*
4. *After the return of the RMI call, the* `Event` *object is returned, thus finishing the* `spawn` *method.*
5. *The method is executed in the SDA and*
6. *the results are communicated back to the caller (more specifically, the associated* `Event`*) using sockets. This terminates the listening thread.*
7. *An activation of the* `getResults` *method of the event assures that all results have been received and*
8. *returns them to the caller.*

■

Requests

As described in Chapter 4 a `Request` object may be passed to the `SDA` constructor as well as to methods `call` and `spawn` instead of an explicit name and explicitly marshaled arguments. `Request` objects are mainly exploited in the context of interfacing HPF components where they are created automatically by the Opus compiler. The implementation of class `SDA` takes care that the contents of request objects is appropriately used in communication with the SDA Server and SDA skeletons.

Request objects not only provide an encapsulation means for requests but also container classes for marshaling results of a method invocation. Recall that the default mechanism is to return the results marshaled within an `Object`. If such a container class is provided both the SDA skeleton and the SDA stub exploit this class for returning the results of a method invocation.

5.3.2 Interfacing HPF with OpusJava

In order to make HPF (Fortran 90) SDAs accessible from within OpusJava the Opus compiler generates stubs and wrappers that route requests from an HPF module to an OpusJava module, and vice versa, and take care of the needed data conversion and marshaling. While the OpusJava framework provides generic interfaces and postpones correctness checks to runtime exploiting

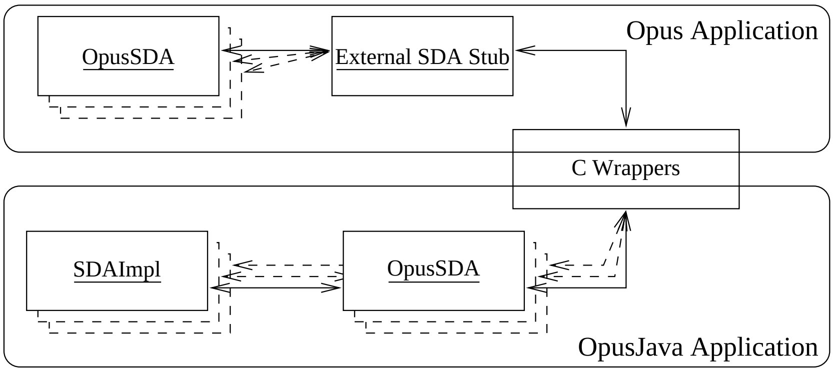


Figure 5.20: Interplay of Opus and OpusJava

Java's reflection mechanism, the Opus compiler requires information about the interfaces of all the SDAs (hence also Java SDAs) in a program at compile time. For this purpose, *Fortran SDA interfaces* designed along the lines of Fortran 90 procedure interfaces [116] have to be provided by the user (cf. Chapter 4).

Based on the SDA definitions as well as SDA interface definitions of an Opus program the Opus compiler generates the following modules:

- A Java counterpart for every HPF SDA type,
- Fortran SDA stubs for every SDA interface,
- an external SDA stub that routes requests from/to HPF to/from the outside,
- C wrappers for interfacing Fortran with Java, and
- an OpusJava support package.

Figure 5.20 illustrates how these components, whose functionality is described below in more detail, work together in an OpusJava application.

Java SDA Classes and Fortran SDA Stubs

For every Opus (HPF) SDA type, a Java counterpart is generated by the Opus compiler. This counterpart is a Java class with the same name and methods as the HPF SDA. A new object of this type is created as an SDA (thus together with an associated skeleton) for every HPF SDA object. Within an OpusJava application methods of these SDAs can be invoked just as of any other OpusJava SDA, however, such requests are transmitted to the respective HPF SDA via C wrappers rather than being served directly. Hence, these Java objects act as OpusJava wrappers for HPF SDAs.

Similarly, SDA interfaces are compiled to SDA stubs that route requests to the matching OpusJava SDA. These stubs act as Fortran wrappers to Java SDAs and are integrated in the External Stub discussed below.

External SDA Stub

An Opus application accessible from OpusJava provides a stub through which all external requests are routed. This stub acts as a multiplexer that transfers OpusJava requests obtained from C wrappers directly to the respective HPF SDA and vice versa. For an Opus application, the stub appears as a normal SDA, thus any request admissible to SDAs can be posed to the stub, as well. But rather than serving the request directly, the stub routes it through C wrappers to the associated OpusJava SDA for service. Note that while every SDA object has its Java counterpart in order to be accessible, only one stub is required for an Opus application. The stub is able to serve all the SDAs of an Opus application. Being an HPF SDA, the stub implicitly takes care of all required gather/scatter operations for implementing data transmissions. Specifically, the stub is an SDA established on a single processor that may communicate with all other (distributed) SDAs by exploiting the advanced communication features described in Section 5.2.4.

C Wrappers

OpusJava defines a generic JNI interface for method invocation requests (thus all the JNI compilation can be done when installing the framework; no additional compilation is needed for particular applications). The Opus compiler generates C wrappers employing this interface which build the link between OpusJava and Opus applications. In particular, the wrappers marshal and unmarshal method arguments and construct the required Opus execution records. The OpusJava `SDA` interface requires method arguments to be passed as an array of `Objects`. This array is transferred to the C wrappers, where it is split up into its components in a way that is conforming to the Opus method signature. In addition, multi-dimensional arrays are packed into appropriate Fortran 90 arrays. The results of a method invocation are marshaled in the same way: multi-dimensional arrays are transformed to Java equivalents and all results are packed into an `Object` (which might be an array in case of multiple output parameters that are possible in Fortran by declaring arguments with intent **OUT** or **INOUT**). This object is eventually returned as result to the Java SDA. Note that the interface only allows Fortran arrays and basic types as parameters. Advanced Fortran features such as derived types are not supported, however, appropriate interfaces are being developed in the ongoing Fortran 200x standardization process and will be employed within our framework once they are available. In addition, serialization or unrolling techniques for derived types can be considered. Initial efforts for unrolling techniques have already been undertaken in an HPF context [132].

The OpusJava Support Package

In addition to the modules that facilitate interaction among HPF and Java SDAs, a Java package (the `OpusJavaSupport` package) is generated which contains supporting Java classes, specifically `Request` classes for all HPF SDAs as well as appropriate `Result` classes. As already discussed in Chapter 4 these classes provide kind of a Java interface to HPF SDAs and may be used to marshal and unmarshal arguments to HPF SDAs appropriately. Although this task could also be performed explicitly by the user, `OpusJavaSupport` classes provide a detailed Java interface of HPF routines and thereby allow static compile time checking of requests.

5.4 Discussion

In this chapter we have presented an implementation design for the Opus system. In particular we discussed the design of a system that efficiently implements the Coordination Language Opus, the internal structure of the OpusJava class library, and the components of the interface among the Coordination Language Opus and OpusJava.

We have identified low level services which have to be available on any target platform. These services specifically 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 for the Coordination Language Opus.

In the exploitation of a restructuring compiler which generates an HPF-conforming code expressing the Opus semantics our approach differs from runtime based approaches like KeLP [72] or HPF/MPI [84] where a runtime library is used to coordinate different tasks. By using a restructuring compiler our implementation is open to future extensions or modifications of the Opus language that might require more advanced compilation techniques. 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. Moreover, the Opus compiler is able to automatically generate all the necessary components required by the interface of HPF to Java.

Based on this design we have implemented prototypes of the Opus compiler, the Opus Runtime System, and the OpusJava class library. We implemented a prototype Opus compiler that accepts a subset of the Coordination Language Opus defined in Chapter 2 and 3 by extending the HPF frontend of VFC [21] and using the Cocktail compiler construction toolbox [95]. The Opus Runtime System is mainly implemented in Fortran 90 and based on SUN MPI [177], *mpich* [94], and pthreads [114]. The data exchange mechanism between distributed SDAs is based upon the adlib [43] library. Prototypes exist for clusters of Solaris Workstations, the Quadrics CS2, and the Cray T3E. Some minor modifications to VFC have also been applied in order to make it interoperable with Opus.

The Java 2 platform is used for the prototype implementation of the OpusJava class library. We implemented a subset of the OpusJava class library and extended the Opus compiler to generate the necessary modules for interfacing HPF with Java as discussed in Section 5.3.2.

Currently, the Opus compiler is tightly integrated with the HPF compiler VFC. However, as we have discussed in Section 5.2.5, our design allows the integration of HPF compilers others than VFC, such as Adaptor [35] or the Portland Group PGHPF compiler [179]. Thereby, mature HPF compilation technologies can be leveraged on any platform. We plan to experiment with these systems in the near future.

Chapter 6

Examples and Experiments

In this chapter we discuss Opus formulations of a number of coordination problems and report on runtime results of a number of experiments. We cover basic coordination schemes, such as barrier synchronization and exchange of signals, as well as complete applications exploiting hybrid parallelism.

We start this chapter with a discussion of Opus formulations of basic coordination schemes: exchange of signals in Section 6.1, barrier synchronization in Section 6.2, the consumer/producer problem in Section 6.3, and the readers/writers problem in Section 6.4.

Then, we present complete formulations of two benchmark programs: fast fourier transform in Section 6.5 and narrowband tracking radar in Section 6.6.

The usage and benefits of lightweight SDAs are exemplified in the following two sections were we discuss a linked list example in Section 6.7 and nested Benders decomposition for dynamic multistage linear programming problems in Section 6.8.

Finally, we present a case study for online monitoring and computational steering that integrates an HPF kernel with a Java interface exploiting the OpusJava framework in Section 6.9.

6.1 Exchange of Signals Between Tasks

In this section, we illustrate how tasks can send signals to each other using SDAs. Assume that there exist two subroutines T_1 and T_2, each being spawned as one asynchronous task. Both tasks want to inform each other about the progress in their execution. Under certain circumstances, a task may have to wait until the other task has reached a given program point.

```
SDA TYPE signal(K,N)
   INTEGER :: K,N
   LOGICAL, PRIVATE :: L(K,N)= .FALSE. ! Signals for K tasks with N labels
CONTAINS
   SUBROUTINE reaching(k,l)           ! task T_k reaches label L(k,l)
      L(k,l) = .TRUE.
   END

   SUBROUTINE wait_for(k,l) WHEN (L(k,l))
   END
END signal
```

Figure 6.1: Exchange of Signals

Let $L_1(i), 1 \leq i \leq N$ and $L_2(i), 1 \leq i \leq N$ denote the sets of labels in T_1 and T_2, respectively. Figure 6.1 specifies the SDA type `signal` for coordinating the two tasks.

Assume s to be an SDA variable associated with `signal`. If task T_1 reaches label $L_1(l)$, it executes `CALL s.reaching(1,l)`. If task T_1 must wait until the other task T_2 reaches $L_2(l)$, then it executes `CALL s.wait_for(2,l)`.

6.2 Barrier Synchronization

Assume that N tasks are to be synchronized at a barrier, where N can be determined at runtime. This can be done by introducing SDA type `barrier`, as specified in Figure 6.2. One complication in this algorithm is caused by the fact that tasks leave the barrier sequentially. We have to avoid that an already released task reaches the barrier again, while there is still another task waiting for release. This is the reason why registering is only allowed if `count < N`.

6.3 Consumer/Producer Problem

The consumer/producer problem represents a class of coordination problems in which one or more independent *producer* tasks compute some values, and one or more independent *consumer* tasks read these values and use them for further processing. The details of producing and consuming are irrelevant for our discussion and ignored. The analysis of the problem focuses on how producers and consumers are coordinated. The assumption made here is that there exists a *bounded buffer* into which all producers write and from which all consumers read. The buffer is managed as a cyclic memory using a FIFO strategy.

The consumer/producer problem is one of the basic coordination problems that occur in practice in many different variants.

More specifically, we formulate the problem as follows:

- Consider a bounded buffer as specified by the SDA type `buffer_type` of Figure 6.3. The buffer is organized as a FIFO queue; writing into the buffer (method `putb`) and reading from the buffer (method `getb`) are atomic actions. In the initial state, all elements of the buffer are *empty*. A call to `putb` consumes one empty element and creates one *full* element; similarly, a call to `getb` consumes one full element and creates one empty element. The condition clauses of these two methods implement the required synchronization conditions: (1) a producer may write into the buffer only if there exists at least one empty element, and (2) a consumer may read from the buffer only if there exists at least one full element.
- There are `nc` ≥ 1 consumer tasks $C(i), 1 \leq i \leq nc$. Each $C(i)$ is a cyclic task, in each iteration reading the next full element from the buffer and then consuming it.
- There are `np` ≥ 1 producer tasks, $P(i), 1 \leq i \leq np$. Each $P(i)$ is a cyclic task, in each iteration producing an element and then writing it into the buffer.
- Apart from the synchronization explained above, all consumer and producer tasks may work asynchronously in parallel.

An Opus formulation of this problem is given in Figure 6.3.

We implemented a simple version of the consumer/producer problem simulating one producer and one consumer acting on a bounded buffer. This simple example was compiled to two versions, one implementing SDAs as threads that communicate via MPI, and the other implementing SDAs as processes as well as threads communicating via TCP/IP. The MPI version was executed on the QSW CS2 and the TCP/IP version was executed on a cluster of Solaris workstations. Figure 6.4(a) shows the execution time for the CS2 and Figure 6.4(b) for the workstation cluster.

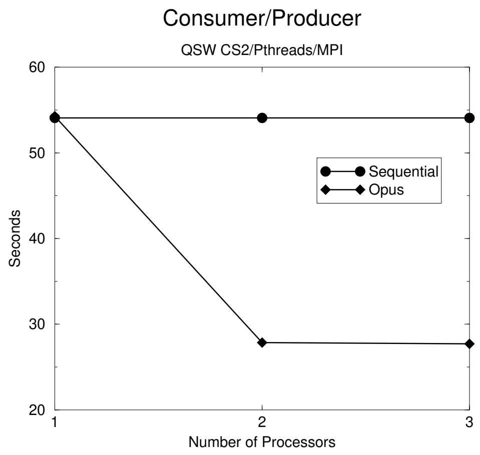
From the results for the CS2 it can be seen that our implementation introduces only a small overhead to the sequential version using one processor. We can achieve almost perfect speedup

```
PROGRAM MAIN
   SDA TYPE barrier_type(N)
      INTEGER :: N                       ! number of tasks
      INTEGER, PRIVATE :: count=0        ! number of tasks reaching the barrier
      INTEGER, PRIVATE :: r=0            ! number of tasks passing the barrier
   CONTAINS
      SUBROUTINE register WHEN (count .LT. N)
         count = count + 1
      END

      SUBROUTINE test WHEN (count .EQ. N)
         r = r + 1
         IF (r .EQ. N) count = 0; r = 0
      END
   END barrier_type

! main program
   SDA(barrier_type) :: barrier
   INTEGER :: num_tasks
     . . .
     ! compute num_tasks
     CALL barrier.CREATE(num_tasks)

     ! generate num_tasks tasks
     DO i = 1, num_tasks
        SPAWN XXX(barrier)
     END DO
     . . .
CONTAINS
   SUBROUTINE XXX(b)
     SDA(barrier_type) :: b
     . . .
! In each task, access to the barrier is represented by the
! following sequence of method calls:
     CALL b.register
     CALL b.test
     . . .
   END SUBROUTINE XXX
END PROGRAM MAIN
```

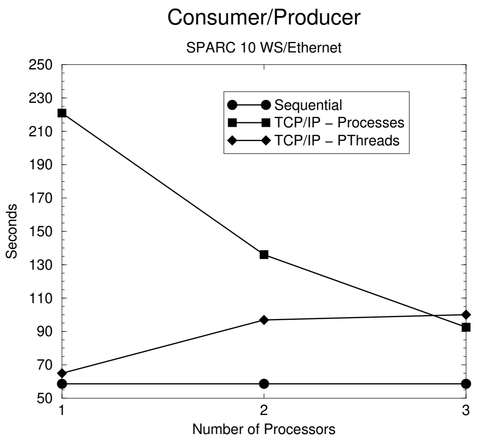
Figure 6.2: Barrier Synchronization

```
PROGRAM Consumer_Producer

   SDA TYPE buffer_type(size)
      INTEGER :: size
      REAL, PRIVATE :: fifo(0:size-1)
      INTEGER, PRIVATE :: count=0     ! number of full elements in fifo
      INTEGER, PRIVATE :: px=0        ! producer index
      INTEGER, PRIVATE :: gx=0        ! consumer index
   CONTAINS
      SUBROUTINE putb(x) WHEN (count .LT. size)
         REAL, INTENT(IN) :: x
         fifo(px) = x                 ! Put x into first empty buffer element
         px = MOD(px+1,size); count = count + 1
      END
      SUBROUTINE getb(x) WHEN (count .GT. 0)
         REAL, INTENT(OUT) :: x
          x = fifo(gx)                ! Read next full buffer element
          gx = MOD(gx+1,size); count = count - 1
      END
   END buffer_type

! main program
   INTEGER :: np, nc, buffersize
   SDA(buffer_type) :: buffer

   READ(np,nc,buffersize)

   CALL buffer.CREATE(buffersize)     ! Create buffer SDA

   DO i=1,np                          ! Spawn producer tasks
      SPAWN produce(buffer)
   END DO

   DO i=1,nc                          ! Spawn consumer tasks
      SPAWN consume(buffer)
   END DO

CONTAINS
   SUBROUTINE produce(b)
      SDA(buffer_type) b
      DO WHILE (.TRUE.)
         ! produce a data item A
         CALL b.putb(A)
      END DO
   END produce

   SUBROUTINE consume(b)
      SDA(buffer_type) b
      DO WHILE (.TRUE.)
         CALL b.getb(A)
         ! consume A
      END DO
   END consume

END PROGRAM
```

Figure 6.3: Consumer/Producer Problem



(a)



(b)

Figure 6.4: Runtime Results of the Consumer Producer Problem

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 CS2 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. These results support our decision from Chapter 5 to implement the Coordination Language Opus based upon a thread-based runtime environment rath than a process-based one.

Figure 6.5 presents the results of an extended producer/consumer example on the CS2. Here, we have executed larger tasks on different numbers of processors where the produce and consume tasks are 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 6.4.

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.

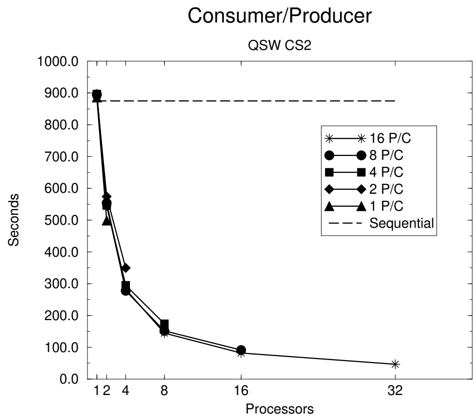


Figure 6.5: Runtime Results of the Extended Consumer Producer Problem

6.4 Readers/Writers Problem

Another common class of coordination problems is represented by the readers/writers problem. It is characterized by a set of one or more independent *readers* and a set of one or more independent *writers* each of which is accessing a common data resource. Writing is exclusive while reading can be done by more than one task in parallel. Moreover, deadlock and starvation must be excluded.

We require the following synchronization conditions to be satisfied:

1. Reader and writer tasks cannot access data resource D at the same time.
2. At most one writer task can access D at any time.
3. If, at the time a writer task terminates execution, there is at least one registered reader task, then all readers registered at this time are given access to the resource.
4. Apart from the previous case, a reader gains access to the resource only if there are no registered writers.

The last two conditions are required to avoid starvation.

We will not go into the details of SDA types, variable and SDA creation for reader and writer tasks, since this can be done along the same lines as for the producers and consumers of the previous example. However, we will discuss the details of the scheduling strategy.

The algorithm is complicated by the fact that starvation must be excluded. The code executed in each reader task and in each writer task is illustrated in Figure 6.6.

Note that access to the resource in readers as well as writers must go through a two-step process. The first step is *registration*: the task simply registers (as a reader or writer, respectively), thus indicating its plan to access the data resource immediately following registration. No blocking can occur in this step. The second step represents the actual entry into the region in which access is performed. This is protected by a condition clause guaranteeing the synchronization conditions specified above.

We will use the following terminology:

- A reader (writer) is *registered* for access to the resource if it has passed registration but not yet completed `begin_read` (or `begin_write`, respectively). The number of registered readers (writers) will respectively be stored in the variables `rr` and `rw`.
- A reader (writer) is *accessing* the resource if it has passed registration and `begin_read` (or `begin_write`, respectively), but has not yet executed `end_read` (or `end_write`, respectively).

```
SUBROUTINE read(d)
   SDA(read_write) d
   . . .
   DO WHILE (.TRUE.)
      . . .
      CALL d.register_read
      CALL d.begin_read
     ! read access to resource
      CALL d.end_read
      . . .
   END DO
END read
```

```
SUBROUTINE write(d)
   SDA(read_write) d
   . . .
   DO WHILE (.TRUE.)
      . . .
      CALL d.register_write
      CALL d.begin_write
     ! write access to resource
      CALL d.end_write
      . . .
   END DO
END write
```

Figure 6.6: Readers/Writers: The Tasks

The number of accessing readers (writers) will respectively be stored in the variables `ar` and `aw`.

- If the situation described under synchronization condition 3 occurs, a writer ending access must release all readers that are registered at this time. This is called the set of *admitted readers*. No writer is allowed to access the resource until all admitted readers have gained access. This is handled via the auxiliary variables `admitted_readers` and `reader_release` and assumes that there is a FIFO strategy for handling a queue of blocked tasks with respect to any specific method call.

The full SDA specification for this synchronization scheme is given in Figure 6.7.

```
SDA TYPE read_write
   INTEGER, PRIVATE  rr=0      ! number of registered readers
   INTEGER, PRIVATE  rw=0      ! number of registered writers
   INTEGER, PRIVATE  ar=0      ! number of accessing readers
   INTEGER, PRIVATE  aw=0      ! number of accessing writers
   INTEGER, PRIVATE  admitted_readers
   LOGICAL, PRIVATE  reader_release=.FALSE.
   . . .
CONTAINS

   SUBROUTINE register_read
      rr = rr+1
   END

   SUBROUTINE begin_read WHEN(aw .EQ. 0 .AND. &
      & (rw .EQ. 0 .OR. reader_release))
      ar = ar+1
      rr = rr-1
      IF (reader_release) THEN
         admitted_readers=admitted_readers-1
         IF (admitted_readers .EQ. 0) reader_release=.FALSE.
      ENDIF
   END

   SUBROUTINE end_read
      ar = ar-1
   END

   SUBROUTINE register_write
      rw = rw+1
   END

   SUBROUTINE begin_write WHEN(aw .EQ. 0 .AND. &
      & (ar .EQ. 0 .AND. .NOT. reader_release))
      aw = 1
      rw = rw-1
   END

   SUBROUTINE end_write
      aw = 0
      IF (rr .GT. 0) THEN
         admitted_readers = rr
         reader_release = .TRUE.
      ENDIF
   END

END read_write
```

Figure 6.7: Readers/Writers: Synchronization

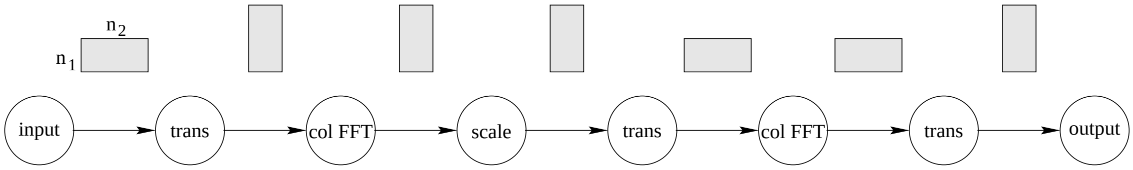


Figure 6.8: Task Graph of the 1D FFT Problem

6.5 Fast Fourier Transform

In this section we discuss an Opus implementation of a 1D Fast Fourier Transform benchmark. This benchmark, and much of its description, is extracted from the CMU Task Parallel Program Suite [66].

The Fast Fourier Transform (FFT) converts an input data set from the temporal/spatial domain to the frequency domain, and vice versa. A discrete Fourier transform (DFT) is a complex matrix-vector product

$$y = F_n x$$

where x and y are complex vectors, and $F_n = (f_{pq})$ is an $n \times n$ matrix such that $f_{pq} = \omega_n^{pq}$, where

$$\omega_n = \cos(2\pi/n) - i \sin(2\pi/n) = e^{-2\pi i/n}$$

and $i = \sqrt{-1}$. A Fast Fourier Transform (FFT) is an efficient procedure for computing the DFT that exploits structure in F_n to compute the DFT matrix-vector product in $O(n \log n)$ time.

If $n = n_1 n_2$, then a 1D FFT can be computed using a collection of smaller independent 1D FFTs. Starting with a view of the input vector x as an $n_1 \times n_2$ array A, stored in column major order, perform n_1 independent n_2-point FFTs on the columns of A. The final result vector y is obtained by concatenating the rows of the resulting array. Figure 6.8 shows the task graph for the 1D FFT.

Input and output are sequences of vectors reshaped as 2D arrays. Nodes labeled *trans* perform a transpose operation, nodes labeled *col FFT* perform a set of 1D FFTs on the columns of its input array, and the node labeled *scale* multiplies each element of its input array by a constant.

We implemented two purely task parallel versions of this benchmark in Opus. Figure 6.9 shows a formulation where the tasks are performed in a pipelined fashion. Because of the relatively small computation time within each of the tasks and the relatively large communication overhead when transferring the array across pipeline stages, we decided to implement a 2-stage pipeline. Hence, we define 2 SDAs, one for each stage. The first SDA contains the *input*, *transpose*, *fft*, and *scale* tasks, while the second one is responsible for the subsequent *transpose*, *fft*, *transpose*, and *output*. In the main loop, the first stage is invoked synchronously and the second one asynchronously. Thus, while the second stage is executing, the first stage is able to process the next input data set. Communication among the two stages is facilitated via array `a` that is passed as output and input argument, respectively.

The second version implements a task-farming scheme. We only define one SDA that takes care of all the steps except *input*. Depending on the number of processors available, a farm of SDA objects, one on each processor, is created. Within the main loop, the input array is read by the master and subsequently transferred to its slaves in a round-robin fashion. The slaves compute on their input sets in an asynchronous manner.

The runtime results for both versions on the QSW CS2 can be found in Figure 6.11. Note that the pipelined version of the code, executed on one processor, adds more overhead to the sequential version than the farming one. This is due to the management of both pipeline stages on one processor, including the data copying when passing the array across stages. Both versions show a good speedup on two processors, however, for this example, exploiting a larger degree of task parallelism does not add much benefit. This is largely due to the communication overhead

```
      PROGRAM FFT
!HPF$ PROCESSORS p(2)
         PARAMETER (n=256)    ! number of columns

         SDA TYPE task1_type
            . . .
         CONTAINS
            SUBROUTINE task1(a)
               REAL, INTENT(OUT) :: a(2,n,n)
               REAL :: b(2,n,n)

               CALL input(b)
               CALL transpose(b,a)
               DO col = 1,n
                  CALL fft(a(:,:,col))
               END DO
               CALL scale(a)
            END SUBROUTINE task1
         END SDA TYPE task1_type

         SDA TYPE task2_type
            . . .
         CONTAINS
            SUBROUTINE task2(a)
               REAL, INTENT(IN) :: a(2,n,n)
               REAL :: b(2,n,n), c(2,n,n)

               CALL transpose(a,b)
               DO col = 1,n
                  CALL fft(b(:,:,col))
               END DO
               CALL transpose(b,c)
               CALL output(c)
            END SUBROUTINE task2
         END SDA TYPE task2_type

         SDA (task1_type) :: t1
         SDA (task2_type) :: t2

         CALL t1.create ON (processors = p(1))
         CALL t2.create ON (processors = p(2))

         DO k = 1,iters           ! main loop
            CALL t1.task1(a)
            SPAWN t2.task2(a)
         END DO
      END PROGRAM
```

Figure 6.9: FFT: Pipelined Version

```
      PROGRAM FFT
!HPF$ PROCESSORS p(number_of_processors())
         PARAMETER (n=256)    ! number of columns

         SDA TYPE fft_type
            . . .
         CONTAINS
            SUBROUTINE compute(a)
               REAL, INTENT(IN) :: a(2,n,n)
               REAL :: b(2,n,n), c(2,n,n)

               CALL transpose(a,b)
               DO col = 1,n
                  CALL fft(b(:,:,col))
               END DO
               CALL scale(b)
               CALL transpose(b,c)
               DO col = 1,n
                  CALL fft(c(:,:,col))
               END DO
               CALL transpose(c,b)
               CALL output(b)
            END SUBROUTINE compute
         END SDA TYPE fft_type

         SDA (fft_type) :: t

         DO i = 1, number_of_processors()
            CALL t.create ON (processors = p(i))
         END DO

         DO k = 1,iters,number_of_processors()  ! main loop
            DO j = 1,number_of_processors()
               CALL input(a)
               SPAWN t.compute(a)
            END DO
         END DO
      END PROGRAM
```

Figure 6.10: FFT: Task Farm Version

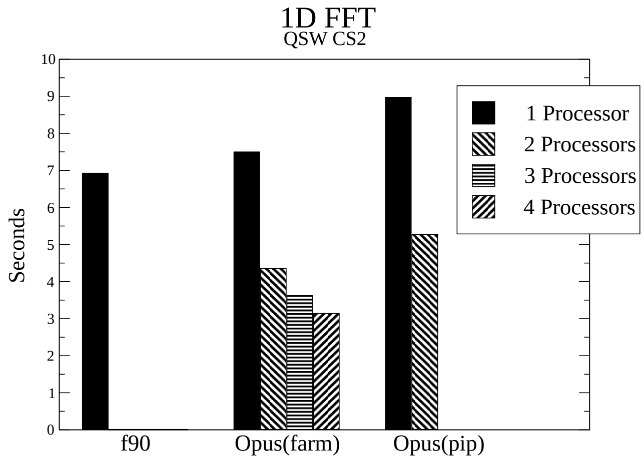


Figure 6.11: Runtime Results of Fast Fourier Transform

among the tasks—remember, that the whole input array needs to be transferred. It is therefore impossible for the master of the task farm to keep all of its slaves busy. However, the FFT may internally exploit data parallelism, thus resulting in a better scaling behavior. The benefits of this hybrid task/data parallel approach can be seen in Section 6.6 where we discuss the Narrowband Tracking Radar problem that is built on top of a data parallel FFT version where the individual FFTs on the columns are executed in parallel.

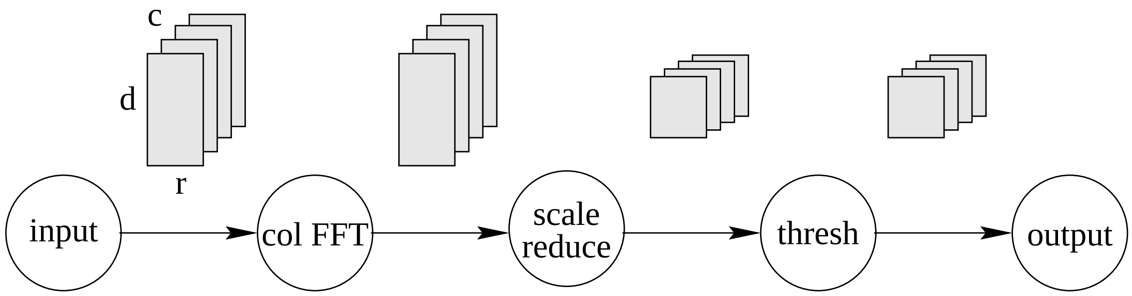


Figure 6.12: Task Graph of the NTR Problem

6.6 Narrowband Tracking Radar

In this section 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 [66]. The Opus implementation was first discussed by Bal and Haines in [10] 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 6.12 illustrates the task graph of the NTR problem.

Figure 6.13 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

```
      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 6.13: The Narrowband Tracking Radar Problem

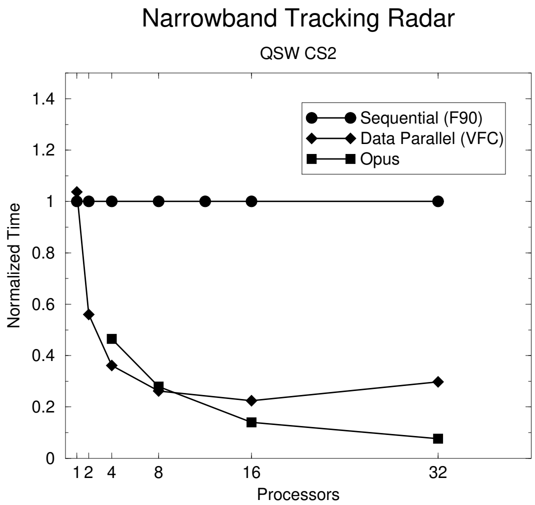


Figure 6.14: Runtime Results of the NTR Problem

allowed to work concurrently on disjoint data sets.

We implemented the NTR problem and compiled it using our prototype Opus compiler. VFC [21] 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 6.14. 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.

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.

```
SDA TYPE node(key,data)
   INTEGER :: key
   TYPE(data_type) :: data

   SDA(node) :: next

CONTAINS

   SUBROUTINE append(n)
      SDA(node), INTENT(IN) :: n

      next = n
   END SUBROUTINE append

   SUBROUTINE retrieve(k,d)
      INTEGER, INTENT(IN) :: k
      TYPE(data_type), INTENT(OUT) :: d

      IF (k .eq. x) THEN
         d = data
      ELSE IF (.NOT. CREATED(next))
         d = null_data
      ELSE
         CALL next.retrieve(k,d)
      END IF
   END SUBROUTINE retrieve
END SDA TYPE node
```

a)

```
SDA TYPE, LIGHTWEIGHT :: node(key,data)
   INTEGER :: key
   TYPE(data_type) :: data

   SDA(node) :: next

CONTAINS

   SUBROUTINE append(n)
      SDA(node), INTENT(IN) :: n

      next = n
   END SUBROUTINE append

   SUBROUTINE retrieve(k,d)
      INTEGER, INTENT(IN) :: k
      TYPE(data_type), INTENT(OUT) :: d

      IF (k .eq. x) THEN
         d = data
      ELSE IF (.NOT. CREATED(next))
         d = null_data
      ELSE
         CALL next.retrieve(k,d)
      END IF
   END SUBROUTINE retrieve
END SDA TYPE node
```

b)

Figure 6.15: Linked List: The Node Declaration

6.7 Linked List

A classical example for lightweight SDAs is the formulation of a LIFO queue of SDAs, implemented as a linked list. Figure 6.15 shows the formulation of a node of such a list as a *heavyweight* SDA (a) and as a *lightweight* SDA (b). Note that the only syntactic difference between these two code versions is the attribute `LIGHTWEIGHT`. Both SDAs define two routines: `append` links a new node and `retrieve` recursively looks for a key in the list and returns the data of the matching node.

Figure 6.16 illustrates the main program for the queue. Note that a (heavyweight) base SDA that builds the root node of the queue is required.

We have executed both versions of the list using different numbers of data items to be stored in the list on a SPARC ULTRA1 workstation. Figure 6.17 shows the runtime results for both versions of the list. We measured the mean time needed for inserting a new element on top of the list as well as the time for retrieving the last element of the list.

While inserting into a lightweight list can almost be done at constant time, the insertion procedure in a heavyweight list involves the spawning of a new thread of control and is thus almost one order of magnitude more expensive. The main advantage of lightweight SDAs can be observed when comparing the execution times for retrieving the last element of the list. While

```
SDA TYPE queue
   SDA (node), PRIVATE :: head
CONTAINS
   SUBROUTINE insert(k,d)
      INTEGER, INTENT(IN) :: k
      TYPE(data_type), INTENT(IN) :: d
      SDA (node) :: help

      IF (.NOT. CREATED(head)) THEN
         CALL head.create(k,d)
      ELSE
         help = head
         CALL head.create(k,d)
         CALL head.append(help)
      END IF
   END SUBROUTINE insert

   SUBROUTINE retrieve(k,d)
      INTEGER, INTENT(IN) :: k
      TYPE(data_type), INTENT(OUT) :: d

      CALL head.retrieve(k,d)
   END SUBROUTINE retrieve
END SDA TYPE queue

! the main program

SDA (queue) :: lili

CALL lili.create()

! insert data
DO key = 1,max_key
   CALL lili.insert(key,data(key))
END DO

! retrieve data
CALL lili.retrieve(1,new_data(1))

END PROGRAM main
```

Figure 6.16: Linked List: The Main Program

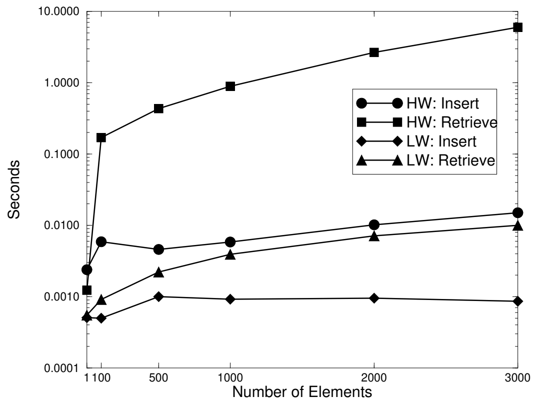


Figure 6.17: Runtime Results of the Linked List

in the lightweight version the retrieval only requires a recursive descent along the list which is executed within one thread, the heavyweight version requires recursive requests through all the threads of the individual list elements. Hence, information retrieval in heavyweight SDA lists is several orders of magnitude slower than in lightweight SDA lists.

6.8 Dynamic Multistage Stochastic Linear Programming

Dynamic multistage stochastic linear programming has many practical applications for problems where decisions have to be made under future uncertainty. The Nested Benders decomposition technique is characterized by splitting the whole problem into a set of small linear programming sub-problems. More specifically, the problem is characterized by an n-ary scenario tree whose nodes are augmented with a sub-problem. Each node receives input from both, its ancestor and its successors. In particular, the ancestor sends its results as new input data while the successors send new constraints on the problem to be solved. For more information on parallel implementations of dynamic multistage stochastic linear programming see for instance [65].

Conceptually, all nodes of the scenario tree work simultaneously on their problems as long as new input from at least one of its successors or from its ancestor is available. There is no other synchronization among the nodes and all communication is assumed to occur asynchronously.

This scenario fits perfectly to the Opus model in that a node of the scenario tree can be implemented as an SDA. However, to model the asynchronous communication among the nodes additional buffer SDAs are necessary to temporarily store the input for a node. This would imply data copies and context switches for every buffer access from within a node. Lightweight SDAs improve this situation in giving the solver running on a node direct access to the data of its communication buffer. In particular, lightweight buffers allow the solver to retrieve new data by means of a standard procedure call instead of an heavyweight SDA method invocation.

The code excerpts given in Figure 6.18 and 6.19 illustrate the Opus formulation. The lightweight SDA `buffer` contains internal storage for buffering both, constraints and results and the respective access functions. Note that a condition clause is associated with `get_data`. This condition clause blocks if no new data is available.

The solver SDA owns SDA variables for its own buffer (`my_buffer`) as well as the buffers of its ancestor and successors (`ancestor_buffer, successor_buffer`). Moreover, SDA variables for the successor nodes are available (`successor`). Note that a reference to the ancestor's buffer is passed as argument to the constructor of the SDA.

Two methods are defined for a node, `init` for initialization of the node and the proper connection to its successors and `solve` which implements the local solver. The `init` routine creates the local buffer and all required successor nodes, passing the local buffer as input argument to the constructor. A reference to the local buffer is returned as a result of the `init` routine such that the ancestor has direct access to it. Subsequently, the `init` method of all the successors is invoked.

The `solve` method initially spawns the solver of all the successors and then enters the optimization loop where in every iteration new constraints and results are retrieved from the local buffer and (if appropriate) the temporal results and constraints are sent asynchronously to the successors' resp. ancestor's temporary buffers.

```
SDA TYPE, LIGHTWEIGHT :: buffer
   TYPE(constraint_type) :: constraints(:)
   TYPE(results_type) :: results
   LOGICAL :: new_constraints = .FALSE.
   LOGICAL :: new_results = .FALSE.

CONTAINS

   SUBROUTINE set_constraints(c,slave_id)
      TYPE(constraint_type), INTENT(IN) :: c
      INTEGER, INTENT(IN) :: slave_id
      constraints(slave_id) = c
      new_constraints = .TRUE.
      ...
   END SUBROUTINE

   SUBROUTINE set_results(r)
      TYPE(results_type), INTENT(IN) :: r
      results = r
      new_results = .TRUE.
      ...
   END SUBROUTINE

   SUBROUTINE get_data(c,r) WHEN(new_constraints .OR. new_results)
      TYPE(constraints_type), INTENT(inout) :: c(:)
      TYPE(results_type), INTENT(inout) :: r

      IF (new_constraints) THEN
         c = constraints
         new_constraints = .FALSE.
      END IF
      IF (new_results) THEN
         r = results
         new_results = .FALSE.
      END IF
   END SUBROUTINE
END SDA TYPE buffer
```

Figure 6.18: DMSLP: The Buffer SDA

```
   SDA TYPE solver(ancestor_buffer)
      SDA (buffer) :: ancestor_buffer
      SDA (buffer) :: my_buffer
      SDA (solver), ALLOCATABLE :: successor(:)
      SDA (buffer), ALLOCATABLE :: successor_buffer(:)
   CONTAINS
      SUBROUTINE init(b)
         SDA (buffer), INTENT(OUT) :: b
         CALL my_buffer.create()
         b = my_buffer

         ! compute number_of_successors

         ALLOCATE(successor(number_of_successors))
         ALLOCATE(successor_buffer(number_of_successors))

         DO i = 1, number_of_successors
            CALL successor(i).create(my_buffer) ON (...)
            CALL successor(i).init(successor_buffer(i))
         END DO
      END SUBROUTINE

      SUBROUTINE solve
         TYPE(constraint_type) :: c(number_of_successors)
         TYPE(constraint_type) :: new_constraint
         TYPE(result_type) :: r, new_result
         ...

         SPAWN successor.solve
         DO
            CALL my_buffer.get_data(c,r)

            ! solve one iteration of the problem

            DO i = 1, number_of_successors
               CALL successor_buffer(i).put_result(new_result)
            END DO
            IF (created(ancestor_buffer)) &
               & CALL ancestor_buffer.put_constraints(new_constraint)

         WHILE (.not. optimal)
      END SUBROUTINE
   END SDA TYPE

! main program

   SDA (solver) :: s
   SDA (buffer) :: null_buffer, root_buffer
   ...
   CALL s.create(null_buffer)
   CALL s.init(root_buffer)
   ...
   CALL s.solve
   ...
```

Figure 6.19: DMSLP: The Solver SDA and Main Program

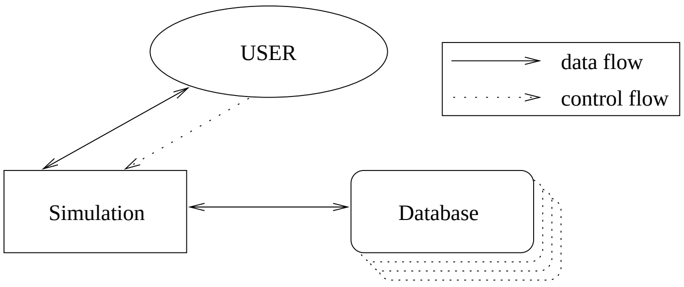


Figure 6.20: Structure of a Typical DHPC Application: Online Monitoring

6.9 Online Monitoring and Computational Steering

In this section we illustrate our OpusJava framework with a prototypic DHPC application: Online Monitoring and Computational Steering [158]. Rather than discussing a specific problem, we present the design of a generic scenario that focuses on the coordination aspects among components. Thus, we can omit specific algorithmic details and illustrate the high level OpusJava programming model in an easily comprehensible way.

Figure 6.20 illustrates the structural design of a typical DHPC application: the user monitors and steers a numerical simulation which receives its data from a distributed database.

It is desirable that the user monitors and steers the application remotely, perhaps using a Java applet. The simulation itself needs to harness massively parallel machines; hence, it is most likely implemented in MPI, HPF, or Opus. The distributed database most likely provides an SQL-like query language.

Such DHPC scenarios are nowadays typically realized by hard-wiring the components and providing problem specific interfaces which are tailored to the application and can hardly be reused. OpusJava, instead, provides a high level component model that gives the programmer an easy means for expressing component interaction, hybrid forms of parallelism, and synchronization at a high level. Our scenario asks for three obvious SDAs: the *user interface* (probably the main program), the *simulation*, and the *database*. In order to decouple the simulation from the user interface to a greater extent, we introduce another SDA, a *buffer*, which is used for storing the current status of the simulation and the simulation parameters, as well.

We assume that the high performance component of our scenario, the *simulation*, is realized with HPF in order to exploit the available resources best. However, the interface to the *database* is probably easier facilitated with the help of a Java wrapper, since Java provides a standardized database interface (JDBC). The *user interface*, that will probably be operated over the Internet is as well implemented in Java. The *buffer* could be realized either with HPF or Java. Since the interaction *simulator/buffer* is more time critical than the interaction *buffer/user interface*, and since method invocation among two HPF SDAs is less time consuming than method invocation among HPF and Java (cf. Figure 6.24), we decided to implement both, the *simulator* and the *buffer* in HPF. Note that the *simulator* might well exploit hybrid parallelism by being composed of multiple data parallel SDAs.

Figure 6.21 shows the declaration of the simulator while Figure 6.22 depicts the declaration of the buffer. The simulator has access to two SDAs, the buffer and the database, and operates a simulation loop that repeatedly retrieves parameters from the buffer, executes one iteration of the simulation, and dumps the current status to the buffer. For sake of brevity we omit the interaction with the database, which is realized in a manner similar to the buffer interaction. The simulation proceeds until the problem converges. The buffer is used to store the status data as well as the parameters and offers 4 methods for accessing its data: `get_parameters` and `set_parameters` for reading/writing the parameters and `get_status` and `set_status` for reading/writing the status

```
SDA TYPE simulator(buffer,database)

  SDA (buffer_type) buffer
  SDA (database_type) database

  ! simulation data
  ! HPF PROCESSOR directives
  ! HPF DISTRIBUTE directives

CONTAINS

  SUBROUTINE simulate

    DO WHILE (.not. converged)
       params = buffer.get_parameters()
       ...
       buffer.set_status(...)
    END DO
  END SUBROUTINE simulate

END SDA TYPE
```

Figure 6.21: Online Monitoring: The HPF Simulator

data, respectively.

Figure 6.23 sketches the Java code for monitoring and steering. We only show the code necessary for interacting with the simulation and omit all the user interface related parts. After the provision of the necessary `Resource` objects (line 4-6), all the SDAs are created: the buffer (7), the database wrapper (8), and the simulator (9). The simulator receives a pointer to the buffer and the database as parameters, hence we use the Opus compiler generated `request` class. Both the simulator and the buffer are allocated on the same machine (machine `M1`) allotting 1 processor to the buffer and `np` processors to the simulator. The database wrapper resides on the physical location of the database (machine `M2`). Subsequently, the simulation parameters are initialized (11) (again exploiting a `request` class) and the simulation is spawned asynchronously (13). The asynchronous method invocation is associated with an event that is used in the monitoring loop (17) for testing whether the simulation has finished. Within that loop, the status is retrieved from the buffer (18), visualized, and, dependent if a tuning of the parameters is necessary, the parameters in the buffer are updated (21). Note that for retrieving the current status a persistent `Request` object is constructed before entering the loop. The compiler generated `Result` class is used for storing the current status.

With our prototype environment we realized the most significant parts of the monitoring application discussed above and measured the time needed for synchronous and asynchronous method invocation between (a) 2 OpusJava SDAs and (b) an OpusJava SDA and an (not distributed) Opus SDA, respectively. We measured round-trip times between two SUN Ultra 10 workstations, running JVM 1.2.1, connected by fast-ethernet. We used different input/output data sets ranging from 0 to 2000 floats[1]. In the synchronous case we measured the time the caller is blocked in the call, while in the asynchronous case the time from initiating the call until the return of an immediately followed `waitEvent` on the associated event was taken. Figure 6.24 shows the mean time elapsed and, for comparison reasons, the pure MPI based Opus/HPF results as well. Note that these timings reflect the full OpusJava mechanisms rather than pure communication times.

[1] We give the array size rather than bytes because of Java's RMI and object serialization mechanisms employed.

```
SDA TYPE buffer_type

   ! status data
   ! parameters
   LOGICAL new_status = .FALSE.

CONTAINS

   FUNCTION get_parameters
      ...
   END FUNCTION

   SUBROUTINE set_parameters(...)
      ...
   END SUBROUTINE

   FUNCTION get_status WHEN (new_status)
      new_status = .FALSE.
      ...
   END FUNCTION

   SUBROUTINE set_status(...)
      new_status = .TRUE.
      ...
   END SUBROUTINE
END SDA TYPE
```

Figure 6.22: Online Monitoring: The Buffer

Moreover, the measurement accuracy for Java is limited to the millisecond range; this may cause some degree of inaccuracy, in particular in the lower range of the measurements.

Comparing the performance of synchronous method invocations in pure OpusJava programs (a) with the reference MPI implementation, we see that without parameters OpusJava is competitive with MPI (subject to the inaccuracy of the Java measurement). However, there is a decrease in performance when more data is being passed. This overhead is mainly due to the well known object serialization overhead of Java. Fortunately, there are already attempts to reduce this overhead [159] and we will study these efforts for our future developments. There is also some improvement potential for Java's RMI mechanism [146] although the performance problems are mainly not due to RMI itself but to the involved object serialization.

This object serialization overhead is also visible in the timings for asynchronous method invocations. However, asynchronous method invocation suffers from an additional performance penalty of approximately 2 milliseconds with respect to the synchronous version. This additional overhead is mainly due to the less efficient thread handling and context switching in the JVM employed. We expect this overhead to decrease with future, optimized JVM releases.

Finally, the timings of the full OpusJava/HPF versions (b) reflect another overhead of approximately 1 millisecond. This overhead is on the one hand due to the additional request handling within the External Stub and on the other hand due to additional data copies that are implied when crossing the language boundaries. Compared to the basic MPI timings we see however that the main overhead is implied by the less efficient JVM while crossing the language boundaries only causes a justifiable overhead.

```
public class Monitor {

   public static void main(String[] args) {
      // create SDAs
      Resource m1 = new Resource("M1",1);  // for the buffer
      Resource m2 = new Resource("M2",1);  // for the database
      Resource m3 = new Resource("M1",np); // for the simulator
      SDA buffer = new SDA("buffer_type",m1);
      SDA database = new SDA("database_type",m2);
      SDA simulator = new SDA(
                     new simulator_request(buffer,database),m3);

      // initialize parameters
      buffer.call(new buffer_type_set_parameters_request(params));

      // spawn the simulation
      Event ev = simulator.spawn("simulate");

      // monitor
      buffer_type_get_status_request r =
                             new buffer_type_get_status_request();
      Result_buffer_type_get_status status;
      while (! ev.testEvent()) {
         status = (Result_buffer_type_get_status) buffer.call(r);

         // visualize and analyze the status

         if (update_necessary)
            buffer.call(new buffer_type_set_parameters_request(params));
       }
   }
}
```

Figure 6.23: Online Monitoring: The Main Program

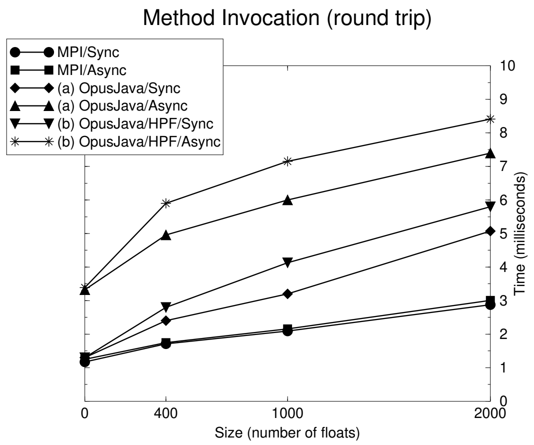


Figure 6.24: Elapsed Time for Method Invocation in OpusJava

Chapter 7

Evaluation of Approaches for DHPC

7.1 Introduction

Distributed High Performance Computing applications as we have described in Chapter 1 exhibit a number of common properties which can be summarized as follows. First, DHPC applications are often composed of software components that are written in different languages and utilize several supporting systems such as numerical libraries. Second, components of DHPC applications exhibit potentials for several (hybrid) forms of parallelism; in particular, coarse grained task parallelism is often available among components while fine grained data parallelism may be exploited within a component. Concurrently executing components need moreover flexible mans for interaction that allow dynamic exchange of information. Third, such applications are expected to be executed in a heterogeneous environment such as a complex cluster.

Programming systems for DHPC applications have to deal with all of these issues. In this chapter we review a number of existing systems with respect to their support for DHPC. We base our discussion on the criteria presented in Chapter 1. Specifically, we investigate how a specific approach deals with heterogeneous hardware architectures (the *heterogeneity* criterion), which interoperability features are offered (the *interoperability* criterion), and finally we examine the features provided for expressing parallelism as well as the respective communication and synchronization means (the *parallelism and communication* criterion).

This chapter is organized as follows: We first develop a classification scheme for categorizing the systems under consideration in Section 7.2. This classification is followed by a detailed description of some selected systems in Section 7.3, focusing on the particular support for the three main criteria named above. Finally, we evaluate the approaches presented in a comparative manner in Section 7.4. Thereby, we contrast them to our Opus approach which we also review with respect to the main criteria for DHPC.

7.2 Classification Scheme

In this section we develop a classification scheme for systems supporting DHPC applications and map some selected approaches which are presented in the following section to this scheme.

Several classifications of distributed and parallel systems exist already: Bal et al. [12] developed a classification scheme for programming languages for distributed computing systems which distinguishes between *distributed address space* and *shared address space* approaches. These classes are further subdivided based on the communication schemes employed. However, their focus is on programming languages—the relatively new field of frameworks and middleware is not covered. Skillicorn and Talia [166] presented a classification of models and languages for parallel computation which is based upon the level of abstraction of parallelism and communication.

In our classification we take a more general approach and distinguish between *language based* and *framework and middleware based* systems. Language based approaches either define a complete language (referred to by *native language approach*) or a set of language extensions (referred to by *language extension approach*) which provide the necessary functionality to express parallelism and communication, either explicitly or implicitly through, for instance, data partitioning. The main advantage of these approaches is that high level language constructs relieve the user from tackling low level details such as message passing and synchronization. Moreover, high level mechanisms are in general easier to use and are less error-prone. However, despite the fact that advanced compilation technology is able to optimize a program to a great extent, high level abstractions may imply performance losses. In addition, many languages exhibit restricted interoperability capabilities and thus do not provide the required flexibility. Portability problems may arise as well if the language is not subjected to standardization.

Frameworks and middleware systems, two relatively new technologies, provide services between heterogeneous platforms and applications [28]. While middleware systems are general-purpose services, frameworks are designed for specific application domains and can thus be regarded as a special kind of middleware. Middleware systems usually only provide means for interaction among components of an application; these components, which may specifically be internally parallel, have to be implemented by means of other languages and systems. The means for interaction provided are often at a lower level than those provided within languages and the user is solely responsible for employing the system correctly and efficiently without the help of an optimizing compiler. Some systems require the user to specify interfaces with an Interface Definition Language (IDL).

In this category we distinguish between *2-tier* and *3-tier* systems. 2-tier systems are typically libraries that provide support for interconnecting software components directly, while 3-tier systems decouple the interaction between software components by means of an additional coordination module. These coordination modules—which may be agents, request brokers, etc. as will be discussed below—usually provide advanced support for heterogeneous distributed environments, however at the expense of an additional system layer. 3-tier systems are often able to provide the user with higher level abstractions than 2-tier systems, because the coordination module is able to cope with higher level abstractions at runtime.

In our survey we discuss a representative selection of all of these classes concentrating on higher level approaches. Low level message passing systems, such as MPI [140, 141], threading systems, such as pthreads [114], or heterogeneous operating systems, such as Legion [92] or Globe [183], are beyond our focus. Similarly, we do not discuss systems that focus on homogeneous architectures, such as Mosix [13]. Our selection is far from being comprehensive and the interested reader is referred to [12, 166] for more in depth surveys. A comparison of approaches explicitly dealing with the integration of task and data parallelism can be found in [10]. The *Grid Computing Environments Working Group* [91] maintains a list of projects related to metacomputing.

In the following, we describe three native languages (Agora, Orca, and Java), four language extensions (HPF, Linda, HPC++, and JavaParty), two 2-tier libraries (KeLP and Globus), and four 3-tier systems (CORBA, WebFlow, NetSolve, and PAWS). We give a short overview of each of these systems and evaluate them against the main criteria presented in Section 7.1.

According to our classification the Opus system is a combination of a language based approach and a middleware system. Specifically, Opus is comprised of the Coordination Language Opus, which is an extension to HPF, and OpusJava, a 2-tier framework. By combining a high level language and a 2-tier framework within a unified system Opus is able to harness the power of either approach.

In our description we use the typographic convention of typing all the keywords in **BOLD UPPER CASE**. Figure 7.1 depicts the mapping of these approaches to our classification scheme.

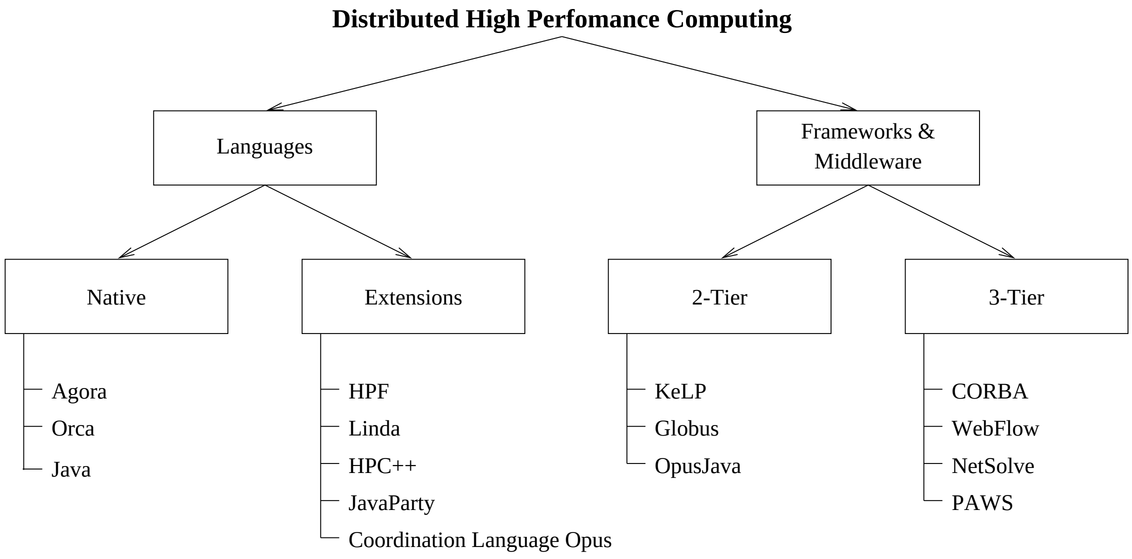


Figure 7.1: Classification of Selected High Level DHPC Approaches

7.3 Selected Approaches

7.3.1 Languages

In this section we present a selection of languages which can be exploited for DHPC applications. Although most of these languages have been designed with other application domains in mind they provide most of the required functionalities for DHPC applications.

We first discuss three native languages, Agora, Orca, and Java. While Agora and Orca both focus on object interaction in distributed systems, Java has been specifically designed as a language for the World Wide Web and has as such gained much recognition.

Instead of defining a new language several proposals have been made to extend existing, well established languages with parallelism and communication features. We discuss four such approaches: High Performance Fortran, a data parallel extension of Fortran, Linda which was one of the first coordination languages that clearly separate coordination and computation matters, High Performance C++, a parallel extension of C++, and JavaParty which introduces transparent remote objects to Java.

7.3.1.1 Native Languages

Agora

Agora [31] is a Lisp-based system that supports the development of multilanguage parallel applications for heterogeneous machines. Agora introduces *shared data types* (SDT) composed of data definitions and access-function definitions. Instances of SDTs, *shared data structures* (SDS) can be addressed either using indexing in a linear-array scheme or through the hashing of strings in a hash table. Agora provides a set of built-in functions for the handling of SDS: **SDS_CREATE**, **SDS_DESTROY**, **SDS_ATOMIC_EXECUTE**, **SDS_READ**, and **SDS_WRITE**. Based on these methods, more complex access functions may be defined by the user.

Computations in Agora are implemented as independent processes that exchange data by using Agora's primitives and transfer control information via *activations*, which are generated as a consequence of a change in an SDS. Shared activation queues are assigned to each process.

Processes specify their interest in an SDS by *events* that contain references to the SDS to which it applies, the process that must be activated, and the activation tag that must be generated whenever the SDS is modified. Events can be installed and removed dynamically by any process.

Agora programs are translated by a compiler to sequential programming languages, such as C or Lisp, supported by communication systems provided at the target machine. The implementation of Agora is based on a single-assignment discipline for variables which eases coherency among copies of shared data, however at the expense of having to copy while updating data structures and imposing the need for garbage collection.

Heterogeneity. Agora explicitly deals with heterogeneity across different platforms. In particular, the data of SDTs is annotated with data type descriptors that allow automatic type conversions. Since both, SDSs as well as processes may be generated dynamically, the hardware properties of heterogeneous environments can be taken into account when setting up the generation scheme. Once created, neither SDSs nor processes can be re-located, though.

Interoperability. Language interoperability in Agora is accomplished by the Agora compiler that translates SDTs to several languages. Initially, Lisp and C were supported. The Agora compiler is thus a cross-compiler that translates SDTs, which are declared using a Lisp-based language, to several other languages. This fact limits the interoperability features of Agora since for any language supported a specific backend has to be developed within the compiler. Apart from this horizontal language interoperability, vertical interoperability as provided by Agora's base languages may be exploited, as well. In particular, an Agora process may well be internally parallel, for instance by exploiting multithreading, however, the user has to deal explicitly with all the details of parallelism and synchronization.

Parallelism and Communication. Parallelism in Agora is exploited through concurrently executing processes which may be started at any time by calling the Agora primitive **NEW_PROCESS**. Thus, Agora employs a coarse grained task parallel discipline where the user has to split the problem into appropriate tasks, implemented as separate processes. Processes communicate with each other by activating methods in a shared SDS. Due to the single-assignment discipline employed, no locks are needed when accessing shared data. However, the user may explicitly force methods of an SDS to execute as an atomic action by means of the **SDS_ATOMIC_EXECUTE** primitive. Processes synchronize with each other by means of *activations* that are generated as result of changes within an SDS. Specifically, a process may employ the primitive **WAIT_ACTIVATION** to suspend itself and wait until it is activated again. Apart from the functional parallelism between processes, no further support for other flavors of parallelism, such as data parallelism, is provided.

Orca

Orca [11] is a language for implementing parallel applications on loosely coupled distributed systems. Sequential processes that are created dynamically on different machines communicate with each other through shared data. This shared data is encapsulated within *abstract data types* (ADT) that encompass data and user-defined operations on the data, similar to SDTs in Agora. Unlike in Opus, Orca objects are passive entities. Parallel execution is expressed through dynamically created processes which may be forked on user specified processors. When forking a new process all shared data need to be passed via parameters of the **FORK** statement.

Orca tasks synchronize through accessing shared ADTs. There are two flavors of synchronization: first, methods of ADTs execute mutually exclusively and, second, sections of ADT methods may be guarded by a logical function, allowing conditional synchronization.

Initially, ADT operations were purely sequential procedures; this restriction was lifted in subsequent developments [18, 19] that introduced *partitioned objects*. Partitioned objects exploit data parallelism within Orca tasks. More specifically, the user may distribute data of a partitioned object over a set of processors and may define *parallel* operations on that data. The execution of these operations follows the *owner-computes* paradigm. If not partitioned, the Orca runtime system may decide to replicate shared objects among all the processes having access to them.

An Orca program is compiled to an ANSI C program that exploits an Orca specific runtime system. This runtime system is based upon a portable runtime support system, Panda [30], which provides basic communication and threading mechanisms.

Heterogeneity. Heterogeneity issues such as different operating systems are tackled by Orca's runtime support system, Panda [30, 165]. The placement of ADTs is dynamic, in particular dynamic ADT migration is provided. Thereby it is tried to achieve locality among shared data and processes accessing this data. Either the user or the runtime system, which is the default case, decide whether to replicate or where to place an ADT initially [9]. Fork statements in Orca may be associated with an **ON**-clause specifying the resources (processors) on which the new processes should be spawned. The numbering of processors is however system dependent, more specifically provided by the underlying runtime system, and the user has thus no means to force an explicit mapping between the virtual processors and the physical hardware. In addition, Orca does not provide means for migrating processes.

Interoperability. Orca does not support the integration of non-Orca components into an application. Thus it neither supports horizontal nor vertical interoperability.

Parallelism and Communication. Orca provides a hybrid task and data parallel model. An Orca program is split into **PROCESSES** which may be forked dynamically as separate tasks. Tasks communicate and synchronize with each other by accessing shared objects which have to be passed as arguments when forking a new task. Orca processes may be forked on more than one processor, thus allowing them to execute internally in a data parallel manner by invoking methods of partitioned objects. Language constructs for specifying the distribution of data within partitioned objects as well as parallel operations are provided. Parallel operations are applied to the data owned by a processor according to the owner-computes-rule. Both, internal objects of a process as well as shared objects may be partitioned. If not partitioned, shared objects may be replicated among all the processes having access to them which on the one hand facilitates efficient read accesses but on the other hand requires reliable (and expensive) broadcast mechanisms when updates occur.

Inter-task communication takes place in Orca as result of invoking methods in shared objects. These methods are executed in a mutual exclusive manner, more specifically, a method execution is an atomic action. In addition, logical conditions may guard the execution of specific code sections. Thus, processes synchronize implicitly by invoking methods in shared objects. In addition to inter-process communication, partitioned objects give raise to communication in case of non-local data accesses. The communication required is handled transparently by the Orca system which applies an all-to-all exchange protocol for replicating non-local data. This expensive data replication may be optimized through static compiler analysis or user supplied hints. Specifically, the user may specify data dependencies that allow the Orca system to perform only partial data exchanges instead of all-to-all. In any case, the necessary communication patterns are determined at runtime.

Java

Java [7] is a general-purpose concurrent object-oriented programming language whose syntax is related to C and C++. Java, originally developed for embedded systems, became the favorite language on the World Wide Web. The main reason for this is Java's platform independence. Java code is not compiled to peculiar machine code, but to platform independent *bytecode* that is interpreted on the target platform by a *Java Virtual Machine* (JVM) [135]. This virtual machine abstracts all the platform-specific features and implements the common API which is employed by the bytecode, using native system calls. Java's bytecode layer allows Java programs to be compiled once and deployed on any platform providing a JVM. Bytecode can furthermore be easily shipped over the Internet facilitating a dynamic deployment of Java programs.

Apart from platform independence, Java provides advanced security features indispensable for programs which are dynamically down-loaded from the Internet. For instance, Java is strongly typed, applies rigorous array-bound checking, and runs a *bytecode verifier* against down-loaded bytecode to ensure correct program behavior. For instance, the code is checked against violations of access restrictions.

Java is an explicitly parallel language in that it provides an built-in low level threads model. Java's threading features are similar to POSIX threads [114], however with a quite different syntax. Moreover, Java supports distributed programming by means of remote objects whose methods can be invoked remotely, as specified in the *Remote Method Invocation (RMI)* specification [175]. As far as high performance computing is concerned, Java has a number of drawbacks, though. First, Java is interpreted which imposes a significant runtime overhead. *Just in time* (JIT) compilation techniques have been developed, such as the HotSpot compiler [6], to speed up time consuming parts of an application dynamically at runtime. Moreover, Java does not provide an efficient array concept (in fact, multidimensional arrays are represented as arrays of arrays) and has some problems with numeric computations due to the data format specification in the language and the floating point arithmetics. An international effort, the JavaGrande Forum [120], tries to identify and eliminate these problems.

Heterogeneity. As already mentioned, architecture independence was one of the major design goals of Java. Any Java program is supposed to run on any machine that provides a JVM. This JVM abstracts from the physical machine and handles all system interaction. Java's RMI model is, however, a static model and remote objects (RMI servers) cannot be re-located after their creation.

Interoperability. Java provides a *Java Native Interface* (JNI) that specifies an interface to C/C++. With help of this interface, procedures written in a native language can be seamlessly integrated into Java programs. JNI allows native procedures to be invoked from within Java, as well as Java methods to be invoked from within native procedures. Java's native interface supports both, horizontal and vertical interoperability. In particular, a native procedure invoked from within Java may, similar to Agora, exploit parallelism through message passing or multithreading.

Parallelism and Communication. Java provides low level shared-memory parallelism as well as a higher level client-server model. Java's built in threading features allow the exploitation of thread parallelism on shared memory architectures while the remote method invocation (RMI) model facilitates distributed programming according to the client-server discipline. Simple point-to-point communication via sockets is available, as well.

In order to exploit Java's distributed computing facilities, remote objects (i.e. objects of classes whose interface extends Java's **REMOTE** interface) have to be registered in a simple name server, the *RMIRegistry*. Another Java program may look-up registered remote objects and invoke their exported methods. Argument transfer is handled by client-side stubs and server-side skeletons which are generated by an RMI compiler that is included within the Java platform. For transmitting argument data Java's object serialization features [174] which allow full objects to be transferred as a stream of bytes are exploited.

Java's multithreading features allow the creation of multiple threads of control that share a common address space within a JVM. These threads communicate via accessing data within the shared address space and synchronize each other by specifying certain code region as **SYNCHRONIZED**—i.e. putting these code regions under the control of a *mutex*, much like in POSIX threads [114].

However, Java is missing a high-level parallelism model, such as SPMD style data parallelism. Moreover, RMI has strict synchronous semantics and does not provide asynchronous method invocations. Several extensions to Java have been proposed which enrich Java with a high level data and/or task parallel programming model [39, 41, 60, 112, 119, 125, 160, 182].

Recently, higher level component and parallelism concepts have been introduced to Java. JavaBeans [103] define a software component model for Java that allows a high level management of coarse grained components as building blocks of an application. JavaSpaces [176] support high level distributed computing by introducing a Linda[1]-like shared object space.

7.3.1.2 Language Extensions

High Performance Fortran (HPF)

HPF [107] is a data parallel extension of Fortran 90 [116] that was designed to provide the user with a high-level interface for programming scientific applications, while delegating to the compiler the task of generating an explicitly parallel (in most cases message-passing) program. HPF has an SPMD execution model and provides the programmer with a logical single thread of control, a global name space, and loosely synchronous parallel computation. The main extensions of HPF to Fortran 90 are *data distribution features*, *data parallel execution features*, and an *extended intrinsic functions and standard library*.

In particular, HPF defines compiler directives for mapping data on arrays of virtual processors (the **TEMPLATE**, **ALIGN**, **DISTRIBUTE** and **PROCESSORS** directives), data parallel directives (such as the **INDEPENDENT** directive which asserts the compiler that a loop is free of loop-carried dependencies and certain side-effects that prevent parallelization), and data parallel language constructs, some of which have been adopted in Fortran 95 [117] (for instance **FORALL** loops).

The first version of HPF [106] was primarily focussed on regular problems and was only limited useful for irregular applications [137]. As a consequence, compilers were sometimes not able to produce efficient code that was competitive with explicit parallel programs for these classes of applications. In HPF-2 several new features have been introduced, in particular irregular data distributions and basic support for task parallelism, however, there is still the need to provide the compiler with better hints about the nature and communication schemes of a program in order to produce more efficient code. Several proposals in this directions have been made [22–25].

Heterogeneity. HPF has been designed for homogeneous, distributed memory architectures. There is little support for explicitly dealing with heterogeneous systems on the language level. Although a compiler may produce code for heterogeneous target architectures, the user has little possibilities of tuning the code taking different architecture properties into account. In particular, HPF does not define a mapping between the virtual processors of an HPF program and the physical processors, thus making adaptive data and work distribution strategies that reflect different properties of target architecture components impossible.

Interoperability. HPF provides an **EXTRINSIC** mechanism that allows to escape from HPF and to call routines written in different languages, such as C, or employing different paradigms, such as explicit message passing, to be called from an HPF program. Hence, HPF has elaborated support for vertical interoperability. Horizontal interoperability is, however, restricted since calling interoperability is only defined for C procedures called from within HPF, but not vice-versa.

Recent developments in the field of high level languages for shared memory architectures, specifically OpenMP [156], stimulated efforts that intend to integrate HPF with these languages in order to obtain a better control over data locality on NUMA SMMPs [34, 56, 57] or to develop a hybrid programming model for clusters of SMMPs [134].

Parallelism and Communication. HPF mainly exploits fine grained data parallelism through parallel loops and array operations. The driving force of parallelism is the distribution of data onto processors. Based on the data distribution, parallel loops and array operations are split into concurrently executing parts by the HPF compiler. Parallelism is detected either by static compiler analysis or specified explicitly by the user (for example the **INDEPENDENT** directive).

Apart from data parallelism, there is some, marginal, support for task parallelism available, in that **PURE** procedures may be called from within independent loops in parallel; in addition, HPF-2 defines *task regions* that are similar to parallel sections found in other languages. However, HPF does not provide means for explicit synchronization and communication among tasks.

[1] Linda is described in detail below.

Therefore, HPF's tasking features are mainly suitable for algorithms which do not require inter-task communication, such as divide and conquer. Several proposals for more advanced tasking facilities in HPF have been made [33, 51, 74, 84, 96, 139, 157].

Inter-process communication in HPF is due to non-local data accesses. An HPF compiler is in duty of generating appropriate code, in most cases message passing code, that takes care of providing non-local data in place. All communication is handled transparently to the user.

Linda

Linda [3, 45] is a coordination language that supports an "uncoupled" style of parallel programming based on a global associatively addressed memory called the *tuple space*. This tuple space can be accessed via atomic operations for adding (**IN**), removing (**OUT**), and reading (**RD**) of tuples. A special **EVAL** operation generates new processes which execute some assigned piece of code and then turn into a tuple that is accessible by other processes in the Linda program. Specifically, one process is generated for each element of the result of the **EVAL** operation. 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.

Linda is a model of process creation and coordination that is orthogonal to the base language in which it is embedded. Language bindings for C, Fortran, C++, Scheme, Modula-2, etc. exist. A Linda system consists of the language-independent runtime kernel, which implements interprocess communication and process management, and a preprocessor or compiler.

Heterogeneity. The tuple space is Linda's key to heterogeneity. It abstracts from a collection of architectures that build a Linda system and provides a uniform interaction model. Moreover, different Linda bindings may interoperate in a heterogeneous system, such that modules may be implemented with a suitable language on any architecture. The implementation of the tuple space in a heterogeneous environment may involve significant overhead, since it requires in essence a coherent shared virtual memory scheme. Based on Linda's tuple space, a system called *Piranha* that provides "adaptive parallelism" on a dynamically changing set of processors, such as in multi-user clusters, has been proposed in [44]. Piranha adopts a simple master-worker discipline where workers are instantiated and re-located dynamically based on the availability of computational resources.

Interoperability. Linda is a *coordination language*, thus it provides a *coordination model* and not a *computation model* [89]. The computation model may be added to Linda by means of a standard base language such as C. Linda's tuple space is the glue that binds separate, heterogeneous activities into an ensemble. Linda provides both, horizontal and vertical interoperability by putting an extra layer of abstraction (the *coordination* layer) on top of the language layer. Linda's coordination layer is particularly useful for coupling and coordinating legacy code.

Parallelism and Communication. Linda provides a coarse grained task parallel model. Concurrently executing processes are generated as result of an activation of the **EVAL** method. Inter-process communication and synchronization is accomplished by inserting and retrieving data from the tuple space. Insertion of tuples is non-blocking while the retrieval of tuples is blocking. Thus, a worker may synchronize with the master by being blocked in a read request for a particular tuple which has not yet been provided. Linda's shared tuple space, which is provided by the runtime kernel, may involve significant overhead, in particular on DMMPs, though.

Apart from coarse grained task parallel programs, Linda may also be used for data parallel programs as is argued in [46]. However, implementing data parallelism with Linda is close to message passing, since Linda lacks high level support for data parallelism and standard data structures. These problems are addressed in more recent work, the *Linda Program Builder* (LPB) [2]. The LPB is a programming environment that supports parallel program construction and parallel data structures at a high level.

High Performance C++ (HPC++)

HPC++ [88] is an extension of C++ [171] for parallel programming which is defined by the High Performance C++ consortium. The design of HPC++ has been partitioned into tow levels. Level 1 consists of a specification for a set of class libraries and tools that do not require any extension to the C++ language. In particular, Level 1 HPC++ comprises three components: a set of simple *loop directives*, a *Parallel Standard Template Library* (PSTL) [121], and a *multidimensional array class*. Level 2 provides the basic language extensions and runtime library needed to implement the full HPC++ Level 1 specification.

HPC++ abstracts from the physical hardware with the notion of sets of interconnected *nodes* which contain several *contexts* consisting of one or multiple *threads*. In this abstraction a node corresponds to an SMP and a context to a virtual address space on a node, typically a process. Based on this abstraction HPC++ provides two execution models. The first is "multi-threaded, shared memory" where the program runs within one context. Parallelism comes from parallel loops and the dynamic creation of threads. The second execution mode is an explicit SPMD model where n copies of the same program run on n different contexts. This SPMD model needs, however, not to be completely homogeneous: multiple contexts may reside on a node and within a context there is the possibility of spawning multiple threads.

However, the definition of HPC++ is still not mature and a complete implementation is currently not available.

Heterogeneity. HPC++ allows the user to tune a program to heterogeneous architectures by allowing arbitrary combinations of contexts and threads on multiple, heterogeneous nodes. Heterogeneity issues such as different type representations are handled transparently by the underlying runtime support systems such as Nexus [83] or Tulip [16].

Interoperability. HPC++ provides low level language interoperability by means of its base language C++. Many languages are interoperable with C/C++, hence HPC++ methods may call procedures written in these languages and vice versa. Similar to Java, this calling interoperability allows to escape from the HPC++ model. An attempt to provide high level interoperation with Java utilizing an adopted subset of Java's RMI mechanism has been described in [36]. HPC++ has also been proposed as runtime layer for a multi-lingual component framework [87]. However, HPC++ falls short as far as vertical interoperability is concerned. HPC++ requires all higher level components that exploit HPC++ as runtime layer to employ its runtime support systems. These systems, specifically the communication library Nexus [83], are often not efficiently interoperable with other communication systems, such as machine specific MPI implementations. This issue will be discussed in more detail in the following description of the Globus toolkit.

Parallelism and Communication. HPC++ provides a data parallel model by means of parallel loops. Parallel loops are specified via HPF-style loop directives that are added via pragmas and comprise **PRIVATE**, **REDUCE**, and **INDEPENDENT**, each of which with similar semantics as the analogous HPF directive. Apart from these directives, SPMD parallelism can be exploited by means of the PSTL which is a parallel extension of C++'s STL. It provides distributed versions of the STL container classes along with versions of the STL algorithms which have been modified to run in parallel. Moreover, parallel iterators and several new algorithms are provided.

In addition to the data parallel model, HPC++ provides a low level shared memory model in that it defines Java-style thread classes for exploiting thread parallelism. High level task parallelism may also be exploited within HPC++. Specifically, HPC++ adopts CC++'s *global pointers* [53] for remote method invocations and provides the collective operations, such as barrier synchronization, available in the multithreading libraries also across multiple contexts.

Communication in HPC++ takes place either transparently within the PSTL or as a result of remote method invocations. However, pack and unpack routines for user-defined classes have to be provided. In order to ease this task, a CORBA IDL to C++ translator that maps interface specifications to user-defined classes, has been developed.

JavaParty

JavaParty [160] introduces "transparent" remote objects to Java. A Java class may be annotated with the class modifier **REMOTE** that allows objects of such a class to act as if they were Java *remote objects*. Given this class modifier, the JavaParty Compiler (JPC) transforms the code into pure Java code with RMI hooks. Thus, methods of JavaParty remote objects are invoked as if through Java RMI. JavaParty, therefore, offers the same features as Java RMI which has been described above. The transformation process of the JPC currently leads to code explosion: from a single remote object ten Java bytecode files are generated.

JavaParty is locality transparent. While remote objects in Java reside on a specific node, i.e. the JVM in which they have been created, JavaParty remote objects are mapped by the compiler and runtime system according to some distribution strategy. This mapping may be guided by the user through appropriate mapping code. In contrast to RMI, communication with remote objects that reside on the same node is optimized in JavaParty and only requires a pointer indirection. In addition, interaction between objects residing on different nodes has been enhanced through an optimization of Java's object serialization and RMI mechanisms [136, 146, 159].

JavaParty supports dynamic migration of remote objects. The decision of migrating an object may be taken by the user explicitly through inserting appropriate migration calls or by the runtime system if migration enhances locality. The user may prevent migration by marking an object as **RESIDENT**, though.

Heterogeneity. Being a Java system, JavaParty yields platform independence. However, there is currently no high-level support for taking heterogeneous platform properties into account, although JavaParty's runtime system, in particular the object distribution and migration facilities, may take hardware properties into account when making placement and/or migration decisions.

Interoperability. Currently, JavaParty is a pure Java system. Although JNI may be used to include native code, the high level concept of remote objects is only applicable to pure Java objects, whereas methods of remote objects in Java may well be wrappers for native procedures.

Parallelism and Communication. JavaParty provides a locality transparent task parallel model at a higher level of abstraction than Java's RMI. Apart from this high level task parallelism, all features available in Java, in particular the built in multithreading, may be exploited within JavaParty, as well. The tasking model of JavaParty is defined analogous to Java's RMI model, in fact, JavaParty harnesses Java's RMI and object serialization mechanisms as supporting tools. Object interaction takes place by means of remote method invocation, hence similar to Java, however in an optimized way.

7.3.2 Frameworks & Middleware

In this section we review a selection of relatively new approaches that exploit runtime technologies instead of language features and associated compilation techniques. With the growing popularity of object oriented software development libraries, frameworks, and middleware systems have been developed that facilitate object interaction in a distributed context. Most of the systems described are specifically designed to support wide-area coordination of distributed applications.

In contrast to most language based approaches, frameworks and middleware systems typically provide language interoperability and are mostly also interoperable with other lower level systems. We will therefore reduce our treatment of the interoperability aspect to a brief discussion of potential problems where appropriate.

We first discuss two runtime libraries: KeLP which provides high level communication abstractions based on a data flow model, and Globus, a toolkit for wide area computing. These systems can be regarded as 2-tier systems since they interconnect application components directly.

3-tier systems which exploit an additional layer for coordination purposes will be discussed afterwards: CORBA, a standardized, industry-strength component model, WebFlow, a graphical component tool, NetSolve, a distributed problem solving environment, and PAWS, a message passing system that focuses on optimizing communication among parallel components.

7.3.2.1 2-Tier Systems

KeLP

Kernel Lattice Parallelism (KeLP) [71, 72] is a framework for implementing portable scientific applications on distributed memory machines. KeLP provides runtime support for general blocked data decompositions and the management of the underlying irregular communication. Parallelism is exploited by means of parallel **FOR_ALL** loops. The KeLP programmer may customize load balancing activity to the application, and treat interprocessor communication at a high level. KeLP encapsulates communication activity using a model known as *communication orchestration*. Under this model, all communication is expressed in terms of atomic array section moves. The implementation of communication follows the *inspector/executor* model [1]. The KeLP programmer expresses and optimizes data motion and decomposition using high-level geometric set operations and may specify elaborate interpolation or coupling functions to handle complicated boundary conditions arising in multidisciplinary applications. KeLP supports a task parallel model which isolates parallel coordination activities from numerical computation. In particular, the KeLP programmer may plug in existing serial numerical kernels that have known numerical properties and leverage mature compiler technology.

While KeLP was originally designed for coordination and communication among sequential computations, an HPF interface (KeLP-HPF [139]) has been developed recently, which allows HPF tasks to be invoked from within KeLP, but not vice-versa. Moreover, support for clusters of SMPs is provided within KeLP2 [8].

Heterogeneity. KeLP is based upon MPI and hence heterogeneous platforms providing a common MPI abstract machine may be harnessed. It is, however, difficult to combine different MPI implementations with each other; a number of projects [17, 68, 115, 168] are dealing with this issue. KeLP2 takes the specific hardware properties of clusters of SMPs into account. More specifically, KeLP2 provides control abstractions that are modeled on the physical properties of SMP clusters: the *collective* level represents the whole cluster, the *node* level an SMP, and the *processor* level processors within an SMP. The dynamic data decomposition features of KeLP enable the user to tune a program's data decomposition for any specific heterogeneous setting that has to provide a common MPI layer, though.

Interoperability. KeLP is a C++ library and may thus be used in conjunction with any language that allows the integration of such a library. In addition, an HPF interface has been developed. Through this interface, data parallel HPF tasks can be coordinated from within KeLP. For this purpose, KeLP's **FOR_ALL** loop has been extended to an **HPF_FOR_ALL** loop that invokes HPF procedures in parallel. However, KeLP does not support interactions of concurrently executing HPF tasks.

Parallelism and Communication. KeLP's programming model is based on a coarse-grain data parallel control flow. Computation in KeLP starts with a single logical thread of control. When entering a **FOR_ALL** loop, loop iterations are processed in parallel on exactly one SPMD process. Communication in KeLP is handled explicitly by the user who makes use of block copy operations between distributed objects. However, no communication may occur within a **FOR_ALL** loop. Similar to HPF's independent loops, the **FOR_ALL** construct may be used to harness some restricted forms of task parallelism.

Globus

Globus [78, 79] is a metacomputing infrastructure 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 seven main modules:

Resource management (GRAM): The Globus Resource Allocation Manager provides mechanisms for expressing application resource requirements, for identifying resources that meet these requirements, and for scheduling resources once they have been located [63].

Communication (Nexus): This component provides basic communication mechanisms including message passing, remote procedure call, distributed shared memory, stream-based, and multicast [82, 83].

Information (MDS): The Globus Metacomputing Directory Service provides a uniform mechanism for obtaining real-time information about metasystem structure and state [73].

Security (GSI): The Globus Security Infrastructure provides basic authentication mechanisms that can be used to validate the identity of both users and resources [81].

Health and status (HBM): The Heartbeat Monitor service provides simple mechanisms for monitoring the health and status of a distributed set of processes and may be used to implement fault tolerance strategies [169].

Remote data access (GASS): High-speed remote access to persistent storage such as parallel file systems is provided by the Global Access to Secondary Storage subsystem [29].

Executable management (GEM): This component is used to initiate computation on a resource once it has been located and allocated. This task includes setting up executables, creating an execution environment, starting an executable, passing arguments, integrating the new process into the rest of the computation, and managing termination and process shutdown.

The Globus strategy is to provide the user with a set of services for metacomputing or Grid applications, rather than with a unified programming model. Therefore, all modules described above contain stand-alone libraries that are designed to work together but may nevertheless be employed separately.

Heterogeneity. Heterogeneity issues are mostly handled by the Nexus communication library that supports data movement across heterogeneous architectures. Moreover, the Metacomputing Directory Service (MDS) provides uniform information about various characteristics of the available hardware and facilitates thus efficient resource allocation and load-balancing strategies.

Interoperability. Globus comprises quite low level libraries that can be integrated in most standard programming languages. However, the user has to deal with issues in integrating parallel modules, such as proper synchronization and efficient communication schemes, explicitly. Moreover, some parts of Globus, such as the Nexus library, prevent the exploitation of similar native libraries on the target architecture. For instance, Nexus can be used to emulate MPI [76], however, it cannot be used in conjunction with native MPI implementations. Hence, Globus provides sufficient support for horizontal interoperability, whereas vertical interoperability is in some cases not supported efficiently enough.

Parallelism and Communication. Globus itself does not provide high level parallelism abstractions. It just contains basic support for dynamic process and thread management and for inter-process communication. The Nexus communication library which is employed within Globus provides basic message passing as well as higher level remote procedure call abstractions and supports communication in a multithreaded context. Nexus also provides an MPI binding such that MPI programs can easily be deployed in a metacomputing context [76]. In addition, Globus supports high-performance I/O through the GASS service which employs optimized data movement strategies for wide area applications.

7.3.2.2 3-Tier Systems

CORBA

The *Object Management Group (OMG)* defined an *Object Management Architecture (OMA)* [20, 184], a framework specification for distributed object-oriented systems. Centered around an *Object Request Broker (ORB)* [155], which is responsible for facilitating communication between clients and objects, the OMA defines four object interface categories:

Object Services [151]: A set of low-level system services necessary for developing applications. These include object persistence, security, transaction support, object naming, and object relationships.

Common Facilities [150]: This is a set of services which are closer to an application level such as document management, electronic mail, and printing.

Domain Interfaces: These are services similar to object services and common facilities, but oriented towards specific application domains such as manufacturing [153], telecommunications [152], or health-care [154].

Application Interfaces: These are interfaces developed specifically for a given application. Because they are application-specific, these interfaces are not standardized, but build upon components of the other three categories.

The *Common Object Request Broker Architecture (CORBA)* [155, 184, 186] is the central part of the OMA. Due to the popularity of the word CORBA, it is commonly used as synonym for all OMA components. CORBA specifies all aspects of client-server interaction where the ORB acts as a mediator between clients asking for a service provided by a server. In particular, a client, having a server object reference, activates an operation which is part of the object's interface. The object reference is opaque and the client has no notion as to the physical location of the object, the language used to implement it, or any implementation-specific details. Once the client issues the request, the ORB is responsible for delivering the request to the object implementation, to invoke the implementation, and to return the results to the client. Object interfaces are defined using the Interface Definition Language (IDL). These interface specifications are on the one hand mapped to a set of programming languages and on the other hand stored in the Interface Repository (IR), from where they can be retrieved in case of dynamic method invocation. IDL compilers translate interface specifications to client-side stubs and server-side skeletons, where stubs are mechanisms that effectively create and issue requests on behalf of a client, while a skeleton is a mechanism that delivers requests to the CORBA object implementation. Object adaptors are used to interface CORBA object implementations with the ORB.

Heterogeneity. The Object Request Broker is CORBA's key to heterogeneity. It hides all architecture specific details and provides a common abstraction from the underlying physical machines. The ORB handles all system interactions and required conversions in a heterogeneous environment. CORBA's task management facility provides support for dynamic re-location of CORBA objects, in particular dynamic agents.

Interoperability. CORBA supports language interoperability through an IDL that allows objects to be implemented in any language for which such an IDL binding is available. Currently, such bindings have been defined for ADA, C, C++, COBOL, Java, and Smalltalk. Vertical interoperability may be exploited as provided by the base language of an object. In particular, low level parallel models, such as message passing and multithreading, may be harnessed in the implementation of CORBA objects. However, CORBA itself has no notion of parallel objects. As a consequence, parallel objects are treated by the ORB as simple sequential ones, and in particular communication among parallel objects takes place through only one channel. Recently, extensions to CORBA for high performance computing, in particular for high performance communication among parallel objects, have been discussed in [85]. These extensions could be implemented as a specialized high-performance CORBA facility component.

CORBA also defines interoperation among different ORBs in the Internet Inter-ORB Protocol (IIOP) as well as interoperation among CORBA and Microsoft's Component Object Model (COM) [20].

Parallelism and Communication. CORBA is an architecture for distributed objects with independent execution of their methods. Thus, CORBA provides a coarse grained task parallel model. In general, object interaction takes place in form of synchronous remote procedure calls. However, CORBA also supports *oneway* and *deferred synchronous* operations which allow concurrent execution of both, client and server methods. More specifically, a oneway operation is a request-only operation that does not yield any results. The caller proceeds immediately after issuing the request and does neither have to wait for results nor to synchronize with the server. In the case of deferred synchronous interaction, the client is allowed to get a response from the server some time after issuing the request. Thus, the client can issue multiple requests in parallel and responses can be collected as they arrive. Apart from this client-server style task parallel model no further parallelism models are available in CORBA itself.

All communication in CORBA, i.e. input arguments and results of remote procedure calls, is handled transparently by the ORB. The client does not know what communication mechanisms the ORB uses to deliver the request to the object and return the response to the client.

Apart from these basic mechanisms, object services for task management are defined. These services include an *event service* for signaling events among objects, and the *concurrency service* that facilitates atomic access to distributed objects. At a high level, task management facilities are defined. These facilities comprise workflow management, support for static and mobile agents, rule management facilities for declarative event-condition-action rule specification and processing, and automation facilities, which are similar to the standard ORB features, but on a higher level of granularity.

WebFlow

WebFlow [104, 105] is a high-level commodity software based visual programming and runtime environment for DHPC. The WebFlow architecture is a three-tier system that comprises a graphical frontend, a CORBA based middle tier and the Globus toolkit as backend. In this sense, WebFlow can be regarded as a visual, high level user interface to Globus.

WebFlow applications are a system of modules, atomic encapsulation units of computation, that communicate by sending objects along channels attached to them. Modules can be dynamically created, connected, scheduled, run, re-located and destroyed. Via a graphical user interface in form of a Java applet the user can request new modules to be initialized, their ports connected, the whole application ran, and finally destroyed. The CORBA based middle-tier comprises the *session manager* that acts like a server for the frontend managing all its requests, the *module manager* in charge of running modules on demand within a separate thread of control, and the *connection manager* in charge of establishing connections between modules. The backend WebFlow modules are CORBA objects, typically implemented as JavaBeans. High performance modules may be integrated in the same way as in other CORBA applications, however, WebFlow specifically supports applications based on the Globus toolkit by providing a dedicated WebFlow module which serves as a proxy of the GRAM (Globus Resource Allocation Manager) gatekeeper client.

Heterogeneity. Through the graphical user interface, the modules of an application may be instantiated dynamically on any resources available to the system. Advanced resources management tools such as Globus' Metacomputing Directory Services are employed. Moreover, the user may dynamically re-locate modules via the graphical user interface. The CORBA based middle-tier is in charge of dealing with all the technical aspects of heterogeneity.

Interoperability. WebFlow's backend modules are implemented as CORBA objects and may thus be written in any language supported by CORBA. Moreover, a specific Globus module is provided that allows the incorporation of parallel Globus applications. Similar to CORBA, parallel modules, exploiting message passing or multithreading, may be included by means of appropriate wrappers. WebFlow also supports the integration of higher level parallel modules, written for example in HPF, however, there is no possibility of dynamic information exchange between concurrently running HPF tasks. In addition, interoperability issues between the HPF runtime system and WebFlow have to be resolved, for instance through employing Globus as HPF runtime system.

Parallelism and Communication. In exploiting CORBA and Globus, WebFlow provides all the parallelism features of these systems. However, while both CORBA and Globus deliver parallelism at a quite low level, WebFlow's graphical user interface allows a higher level specification of task parallelism according to the data flow paradigm. Communication among WebFlow modules is specified graphically by appropriate communication links. At the time a module is instantiated, all necessary communication links are set up and standard CORBA protocols are used for communication.

NetSolve

NetSolve [47, 49] is a client-server application designed to solve computational science problems over a network. NetSolve allows the use of network resources for numerical computing by providing convenient access to problem-solving techniques (software) in addition to managing computational resources. With the help of NetSolve remote software and computation resources can be seamlessly integrated into applications.

The NetSolve architecture comprises three components:

- A set of *client APIs* that allow functions defined in various numerical libraries to be called as remote procedure calls from programs written in MATLAB, C, Fortran, or Java;
- a set of *servers* that encapsulate remotely accessible numerical libraries; and
- one or more *agents* that match requests for services with servers.

The main difference between NetSolve and classical client-server architectures is the employment of intelligent *agents* [50]. Agents are processes that reason about and make decision about the mapping of computations to resources. Hence, agents take scheduling decisions such that a given problem will be solved with the most suitable software on the most suitable system. Moreover, agents are able to tackle fault tolerance issues.

Heterogeneity. By its nature, NetSolve is a heterogeneous system. However, all heterogeneity issues are handled transparently by NetSolve's agents. They take care of selecting the most suitable architecture for a given problem and handle data format conversion, for which the XDR protocol [172] is exploited. Agents facilitate task migration in case of performance degradation by exploiting checkpoint-restart mechanisms.

Interoperability. NetSolve currently provides an interface to MATLAB, C, Fortran, and Java. Moreover, NetSolve server functions may be written in any language and exploit any form of parallelism, as long as they can be described by means of NetSolve's description language. Hence, NetSolve transparently integrates coarse grained task parallelism (asynchronous function calls) with any form of parallelism exploited within the functions. However, NetSolve employs strict remote procedure call semantics: there is no possibility of interaction among NetSolve server functions called concurrently from a client.

Parallelism and Communication. Parallelism in NetSolve is exploited at several levels. First, libraries offered by NetSolve may be parallel. NetSolve makes use of a machine-independent *description language* to describe functions offered by computational servers. These descriptions may even be generated automatically by a Java applet [48]. Second, NetSolve provides *synchronous* and *asynchronous* calling mechanisms on the client side. Asynchronous NetSolve calls can be executed in parallel and agents take care to schedule them accordingly. NetSolve clients communicate with servers by passing input arguments to NetSolve functions and receiving results. Currently, all communication is based upon TCP/IP [113] and handled transparently by NetSolve's agents.

PAWS

The *Parallel Application WorkSpace* (PAWS) [15] is a software infrastructure for use in connecting separate parallel applications within a component-like model. PAWS provides in particular means for dynamically coupling of applications and supports efficient communication of distributed data structures. The PAWS architecture comprises the PAWS *Controller* which coordinates the coupling of applications, manages resources, and handles user authentication in a similar way as an ORB, and PAWS *Applications* which communicate with the controller via the PAWS API. The implementation of PAWS builds upon the Nexus [82, 83] library (see the discussion of Globus above).

Heterogeneity. Heterogeneity issues in PAWS are handled by the underlying Nexus library. Moreover, PAWS takes properties of parallel applications, such as the number of processes, and hardware characteristics, such as network topologies, into account when generating communication schedules. PAWS does not support re-location of running applications, though.

Interoperability. Currently, PAWS is a C++ library; C and Fortran interfaces are under development. Applications written in any language that allows to incorporate such libraries can be interconnected with PAWS and may communicate exploiting the common PAWS layer. However, PAWS does not support interoperation with high level parallel languages and the user is responsible for providing a detailed data description via an appropriate PawsData object.

Parallelism and Communication. PAWS is designed to coordinate a parallel execution of multiple, interconnected programs. In that respect, PAWS has similar goals as CORBA, discussed above, and employs a similar strategy based on the PAWS controller, which is comparable to an ORB. PAWS supports a coarse grained task parallel model in that full applications may either register themselves within the PAWS controller or may be directly started under to control of the PAWS controller. Although PAWS does not provide means for other forms of parallelism, it assumes that the controlled applications exploit some form of parallelism and PAWS puts a lot of effort in optimizing communication between parallel applications. Instead of serializing messages between parallel applications, as done in CORBA, multiple communication channels are exploited. In order to employ optimized communication schedules, PAWS requires information about the layout, the location, and the storage type of the data, all of which has to be provided by the user through appropriate *PawsData* objects. PAWS communication may take place *synchronously* or *asynchronously*.

7.4 Evaluation

Figure 7.2 summarizes the features of the approaches discussed in Section 7.3 as well as of our Opus approach with respect to their support for heterogeneity, interoperability, task parallelism, data parallelism, and the communication means available (shared address space, shared objects, message passing). A "+" indicates that the respective feature is supported by an approach, while a "−+" indicates that it is not fully supported or that there are some open problems. Features marked with "−" are not supported.

Heterogeneity

All of the approaches presented provide some basic functionality that allows them to work in heterogeneous environments which is a crucial requirement for DHPC. In general, heterogeneity is hidden by a virtual machine that is provided either by a message passing system (such as MPI, onto which most HPF compilers target), an interpreter (such as the JVM in Java), or some kind of coordinator (such as CORBA's ORB).

Apart from the basic requirement of being able to execute in a heterogeneous environment, active support for efficient exploitation of the available architecture is crucial. More specifically, means for dynamic task creation as well as task re-location are of importance. High level languages such as Orca, HPF, and Java provide only modest support in this respect. Orca allows to create tasks dynamically, which can be done according to some load-balancing scheme, however, once created, Orca tasks cannot be re-located; the same holds for Java, whose RMI mechanism implements a static client/server model. In contrast to these approaches Opus supports both, dynamic object creation as well as object re-location. However, all placement decisions have to be

Approach	H	I	TP	DP	Communication SA	SO	MP
Agora	+	−+	+	−	−	+	−
Orca	−+	−	+	+	−	+	−
Java	+	+	+	−	−	+	+
HPF	−+	−+	−+	+	+	−	−
Linda	+	+	+	−	−	+	−
HPC++	+	−+	+	−+	−	+	−
JavaParty	−+	−+	+	−	−	+	−
KeLP	−+	+	−+	−+	−	−	+
Globus	+	−+	+	−	−	+	+
CORBA	+	+	+	−	−	+	+
WebFlow	+	+	+	−	−	+	−
NetSolve	+	+	+	−	−	+	−
PAWS	+	+	+	−	−	−	+
Opus	+	+	+	+	−	+	−

Figure 7.2: Summary of Selected DHPC Approaches

(H = Heterogeneity; I = Interoperation; TP = Task Parallelism; DP = Data Parallelism; SA = Shared Address Space; SO = Shared Objects; MP = Message Passing)

taken explicitly by the user and there is currently no automatic support for this tasks. Advanced automated mapping support is available in JavaParty which extends Java with object distribution and migration facilities. The JavaParty runtime system dynamically optimizes object mapping, however without taking hardware properties into account. In addition, the user may explicitly enforce mapping decisions. HPF and its dialects provide advanced features for dynamic and irregular data distribution, however, data distribution targets virtual processors and there is no mapping defined among virtual and physical processors. Thus HPF is able to cope with irregularities within applications, but assumes a homogeneous hardware structure. Opus, instead, allows to explicitly specify the mapping of HPF tasks onto hardware components thus providing active support for heterogeneous architectures. For each HPF task the standard HPF semantics apply, thus it assumes a homogeneous architecture, though. Finally, Linda's tuple space can be used to implement several load balancing schemes and is also used in the Piranha system that supports dynamic task migration.

Agent based middleware systems, such as NetSolve, provide advanced support for heterogeneity. The agent deals with different architectures completely transparent to the user and is also able to employ several load balancing strategies. Similar services are provided by CORBA's task management facilities. Globus provides these functionalities as well, however a lower level: the user is required to implement load balancing strategies exploiting components of the Globus toolkit, such as MDS. OpusJava provides the same migration mechanism as the Opus language. In particular, an OpusJava SDA offers a **MIGRATE** method that allows to trigger object migration. However, the user is solely responsible for employing this feature; there is currently no system support as in other middleware systems. Nevertheless, supporting systems, such as those offered by Globus, could easily be integrated within OpusJava due to its open design. See Chapter 8 for a more detailed discussion.

Interoperability

Horizontal interoperability is typically provided by middleware systems that are often designed to coordinate several application components independently of the respective language they are implemented in. Some additional effort is often required for specifying appropriate interfaces, for instance through CORBA's IDL. These elaborated language interoperability facilities are particularly valuable in integrating and coordinating legacy software modules. However, middleware systems are often not able to peacefully co-exist with third-party systems—vertical interoperability often requires the low level system to be emulated by the higher level framework. OpusJava, instead, does not require any specific supporting system and is therefore able to co-exist with any system that is interoperable with Java. In addition, OpusJava allows a seamless integration of HPF modules within a Java application. Similar functionalities are offered for instance within WebFlow, however, WebFlow requires HPF modules to be based upon Globus whereas OpusJava is interoperable with any HPF system supported by the Opus runtime system.

While middleware systems provide good language interoperability, this feature is often quite restricted in language based approaches. Java, for instance, only provides an interface to C, but with full calling interoperability in both ways. This calling interoperability is not present in HPF. The extrinsic mechanism of HPF allows to escape from HPF to other languages or paradigms but it is not possible to call HPF procedures from such extrinsic modules. In the language category, Linda exhibits the best interoperability features since it is a pure coordination language that is able to handle computation modules written in any language for which a Linda binding has been specified. Similar approaches, on a lower level, are pursued by some middleware systems that require any supported language to define a mapping to some IDL. Agora, finally, provides language interoperability via a cross-compiler. This approach is quite restricted since fully fledged backends need to be developed; moreover, cross-compilation across paradigm boundaries, such as object oriented vs. procedural, is a non-trivial task. The Coordination Language Opus extends HPF with both way calling interoperability, in particular through its integration with OpusJava. Moreover, different HPF systems may peacefully co-exist within Opus applications as long as they are supported by the Opus runtime system.

Parallelism and Communication

Task Parallelism. As can be seen from the summary all of the approaches discussed provide some support for task parallelism. This fact is not surprising, since the possibility of coupling several application modules in a coarse grained task parallel manner is essential for any system supporting DHPC applications. Taking a closer look at the task parallel features provided we recognize that many systems support client/server style interaction or a strict data flow paradigm, but not interaction among concurrently executing components. Lack of this flexibility is a crucial weakness with respect to DHPC.

Client/server style task parallelism is the dominant model employed in frameworks and middleware systems. Although this discipline is applicable to a variety of problems, more general systems are required for DHPC applications, in particular systems that provide dynamic interaction among concurrently executing tasks. This functionality is only provided by low level systems, such as Globus, where explicit message passing may be used to exchange information. The data flow paradigm, under which data flows from a producing task to a consuming task, is employed for example in KeLP and WebFlow, the latter one providing a graphical programming model. In these schemes, dynamic interaction among concurrent tasks is not possible.

Only modest task parallel support is provided by HPF. One of the design requirements of HPF was the provision of a single logical thread of control. This requirement was also obeyed in the design for task parallelism resulting in lack of flexibility making even dynamic master/worker schemes inexpressible. Direct communication among concurrent HPF tasks by means of send/receive primitives, respectively an MPI binding for HPF, has been proposed in [33, 84]. Although these systems allow more efficient task interaction they require the use of low level message passing primitives within a high level language. More flexible means are provided by the other languages discussed, Agora, Linda, Java and its derivatives, and Orca. All of these approaches provide some sort of shared objects or a shared object space that can be used to exchange information among concurrently executing tasks. Java's model is not as elaborated as the other ones, since its RMI mechanism only allows synchronous method invocations. However, with help of multithreading techniques the user may circumvent this problem and recent developments, such as JavaSpaces, provide more flexible means. Shared objects are valuable means for dynamic task interaction since they decouple senders and receivers and provide kind of a message buffer. Hence, asynchronous and even non-deterministic messages are easily expressible. The shared object model is also employed within Opus. Both, the Coordination Language Opus and OpusJava are based on ShareD Abstractions that allow the exploitation of flexible task parallelism by providing synchronous as well as asynchronous method invocation together with high level synchronization means.

Data and Hybrid Parallelism. While some form of coarse grained task parallelism is provided by all of the approaches discussed, high level data parallel features are only available in some, mostly language based, approaches. Obviously, frameworks and middleware systems often cannot provide high level means for data parallelism, since such features would in most cases imply associated compiler development. Hence, data parallelism may be implemented either directly by using low level means such as message passing or the systems allow an integration of parallel modules. However, if parallel modules are integrated into a framework the framework is often not aware of the internal parallelism and hence not able to optimize interactions among data parallel components. In particular, inter-component messages have to be sequentialized, in most cases by some kind of gather/scatter mechanism, instead of exploiting multiple communication channels in parallel. KeLP and PAWS are two systems that take the parallel structure of components into account and generate optimized communication schedules. However, only KeLP supports an integration of high level data parallel languages, in particular HPF. The integration of HPF within KeLP requires much user interaction, though, whereas OpusJava allows a seamless integration of HPF modules: all data marshaling issues are taken care of by the Opus compiler.

In the language category, HPF and Orca provide a high level data parallel model based upon data partitioning and mostly data driven work distribution. Indeed, HPF was designed as a data parallel language, while Orca originally was a task parallel language that has been extended with data parallel features later on. With this history in mind, it becomes understandable that HPF provides elaborated data parallel features, but only modest task parallelism, while Orca's task parallel model is quite flexible, but its data parallel model is not as sophisticated as that of HPF. The Coordination Language Opus extends HPF with a valuable task parallel model thus providing high level integrated support for both, task and data parallelism. HPC++ also provides some support for data parallelism, in particular directives for specifying parallel loops, however it lacks high level data partitioning means and has not yet been fully specified.

Communication. The lack of support for hybrid parallelism in most systems becomes also evident in evaluating the communication features provided. Most of the approaches discussed provide communication by means of accessing shared objects. In particular, communication takes place in flavors of remote procedure calls where input data as well as results have to be transferred. While in a purely task parallel environment this data transfer is typically handled transparently, interaction between parallel components becomes more complicated. In particular, the user is required to deal explicitly with distributed data and even to manage gather/scatter operations that are necessary because of the serialized communication among parallel components. Instead of using shared objects for data exchange, Java, Globus, KeLP, and PAWS also provide (additional) message passing abstractions. While Java only provides simple socket based communication, however with a comfortable object transfer mechanism—object serialization, the Nexus library included in Globus offers an MPI interface. KeLP provides higher level block copy operations and PAWS specifically deals with optimizing communication of distributed data; the user is required to specify the data layout explicitly, though.

High level languages that provide a task and a data parallel model, such as HPF and Orca, employ sophisticated compilation and runtime strategies for facilitating communication among parallel components. However, HPF does not allow direct communication between tasks and Orca's data parallel model is restricted. Opus provides communication facilities among HPF tasks by means of elaborated data redistribution libraries. Also, gather/scatter mechanisms required for communication among HPF and OpusJava objects are provided implicitly by the Opus compiler.

Thus, Opus is, apart from Orca whose data parallel model is restricted, the only system that provides an integrated, high level hybrid parallel model together with efficient communication means.

Summary

Our evaluation of the approaches selected clearly reveals a lack of high level support for hybrid parallelism. Either sophisticated support for data parallelism or task parallelism is provided, but a sufficiently flexible and efficient combination of these two paradigms is currently hardly possible—neither on a low level nor in high level languages. The Opus system is, apart from Orca, the first attempt to provide the required hybrid parallel model at a high level, thus relieving the user from tackling low level communication and data marshaling details. Orca, which offers similar parallelism features, lacks however adequate support for hybrid systems and interoperability and is therefore not an appropriate option for DHPC.

Currently, middleware systems provide the best support for heterogeneity in architectures and languages. However, performance problems may arise due to the generality of these systems. This is in particular true for 3-tier systems. Performance problems also arise in the Java language, due to the interpretation of bytecode at runtime. Nevertheless, Java is a valuable tool for coordinating components in heterogeneous environments and as such the basis for our OpusJava framework. However, the Opus system including OpusJava only provides minimal support for heterogeneous environments and lacks in particular active system support for devising efficient mapping strategies on heterogeneous architectures.

Another open issue that is not sufficiently supported by most current systems are efficient and easy means of integrating native libraries, such as numerical or even message passing libraries. This vertical interoperability feature is inevitable for achieving high performance. In developing our Opus system we put much effort in an open modular design that does not hinder the integration of systems others than employed within Opus.

The standardization of (parts of) a system is also an important issue. Without sufficient standardization, the availability of a system may be reduced and, most important, industry strength implementations will hardly appear on the market. Currently, HPF, Java, and CORBA are standardized and industry strength implementations of all of these three systems are available. In developing our Opus system we tried to employ standardized components as much as possible. As a consequence the Coordination Language Opus is an extension to HPF and its runtime system is comprised solely of standardized components such as MPI and pthreads. Similarly, OpusJava is a pure Java framework that can be employed on any standard Java machine.

In selecting the appropriate tool, designers of DHPC applications have, for the time being, to trade off between flexible support for heterogeneous component programming and high performance. The development of efficient, high level, hybrid parallel models is therefore of paramount importance. As a valuable tool for DHPC we envision a system that combines high level hybrid parallelism, high performance, and flexible distributed computing means in a coherent framework. We believe that our Opus system is a first attempt to provide a useful and efficient tool for DHPC application developers. By being comprised of both a high level language and a 2-tier framework, both of which are tightly integrated, the user is able to select the appropriate tool for any component of an application within a unified system. The primary effort in the development of Opus was put in the provision of an integrated high level hybrid parallel model as well as in interoperability issues. Basic support for heterogeneous architectures is provided by Opus as well, however other issues in DHPC such as security, accounting, and fault-tolerance are not tackled, yet (cf. Chapter 8).

Chapter 8

Conclusions

In this thesis we have presented the Opus system, a high level object based component system for distributed high performance computing (DHPC). Opus facilitates a high level coordination of data parallel tasks in multi-lingual heterogeneous environments. In particular, the Opus system consists of the Coordination Language Opus providing an integrated task and data parallel model, and OpusJava, a Java framework for DHPC that integrates seamlessly with the Opus language. The key concept of Opus is the ShareD Abstraction (SDA), an abstract data type that encapsulates distributed data and data parallel procedures. SDAs provide high level task coordination means and build the common basis for interoperation among different languages. Furthermore, we have designed compilation and runtime strategies that efficiently implement the Opus constructs on a wide range of distributed and parallel architectures. Prototype systems have been developed to evaluate the efficiency of our design. Finally, we gave an in-depth survey of selected approaches that may be applied for DHPC focusing on their support for three major issues in DHPC: heterogeneity, interoperability, and hybrid parallelism.

8.1 Contributions

The primary contributions of this thesis are summarized below.

- *Extensions of the Opus language*
  We have refined and extended the Opus language [55] to meet the key requirements of DHPC. These extensions include:
  - The SDA concept has been refined to provide more flexibility with respect to SDA creation, termination, and synchronization. Specifically, we introduced new intrinsic methods for SDA creation status inquiries (**CREATED**), immediate termination of SDAs (**KILL**), and nondeterministic synchronization (**WAITANY**).
  - A compiler directive called ParBlock has been introduced to allow a concurrent execution of methods within an SDA, thereby relaxing the standard monitor semantics of SDAs.
  - Standard SDAs have been designed to be active objects that act as computation as well as intelligent data servers. Additionally, we introduced lightweight SDAs that are passive objects with reduced creation and method invocation overhead with respect to standard SDAs.
  - A key requirement of DHPC applications is that they ask for dynamic adoption of the distribution of their components. A new intrinsic method **MIGRATE** has been defined that allows a dynamic re-location of SDA objects during program run.
  - Finally, a priority clause has been introduced that provides some control over the order method invocation requests are served by an SDA.

- *Specification of the OpusJava framework*
  A Java binding of Opus, called OpusJava, has been designed that is interoperable with the Opus language. OpusJava may be employed for making Java and HPF components of an application interoperable, as a high level component framework for distributed Java applications, and as machine independent communication layer for the Opus language.

- *Implementation strategies for the Opus language*
  We developed a design for an implementation of the Opus language that is portable to a wide range of machines without sacrificing high performance. Through a modular design components of the system can be easily adopted to specific requirements of particular target architectures. We presented a global view of the Opus system ranging from an abstract execution model up to the high level compiler component. The functionalities provided by the language specific runtime system as well as the system support required from any target architecture have also been discussed in detail. The system was designed to work in close cooperation with an HPF compiler that performs the translation of the data parallel parts of Opus. Although we considered a specific HPF compiler (VFC) there is nothing inherently in our design that prevents the incorporation of other HPF compilers.

- *Implementation strategies for OpusJava*
  We specified the runtime behavior and the API of system classes needed to implement the OpusJava API. The OpusJava framework has been designed as a pure Java package.

- *Integration of multi-lingual components within an application*
  A Java/HPF interface has been specified as an integrated part of OpusJava. We described wrapper functions required for data marshaling. Moreover, Fortran interfaces and a Java package for inter-language method invocation requests have been specified.

- *Prototype systems*
  The efficiency of our implementation design has been evaluated by means of prototype implementations of various system components. Specifically, we implemented the Opus runtime system on a range of machines, including the QSW CS-2, the Cray T3E, and clusters of Solaris workstations. An Opus compiler that accepts most of the basic Opus language constructs has been developed by extending an existing HPF compiler—VFC. This compiler also generates the necessary interfaces to OpusJava. The OpusJava framework has been implemented using the Java 2 platform. A number of experiments have been conducted with this prototype environment.

8.2 Future Research

The Opus system as presented in this thesis is a first attempt to provide an integrated environment for DHPC. In our development we focused specifically on an efficient and flexible integration of task and data parallelism as well as on language interoperability issues. Several other important issues, such as heterogeneity, security, and fault tolerance, have not or only partially been tackled. In the following we summarize the most important research directions that will be examined in the future.

- *Enhanced object placement*
  Opus allows dynamic object placement and migration where the user may specify the resources available to an SDA by means of an **ON**-clause (in the Opus language) and a `Resource` object (in OpusJava), respectively. With these mechanisms the user is able to

name a machine and specify the number of processors that are available to an SDA. However, Opus lacks support for systematically categorizing components of complex clusters. JavaSymphony [69], for instance, provides high level means for specifying abstract topologies on top of cluster systems in conjunction with static and dynamic system properties such as CPU load, available memory, network bandwidth, etc. Similar features are provided by the Globus resource management system [63] that comprises intelligent resource brokers, resource managers local to physical machines, and an information service that provides information about the current capabilities of resources. We will examine these systems and incorporate suitable parts of them in our Opus system.

- *Scheduling support*
  A key issue for achieving high performance in DHPC is an efficient scheduling of application modules onto the components of the target complex cluster. The scheduling decision must take into account not only application properties but also system properties; moreover, dynamic re-scheduling might be necessary. An example of a system that considers both properties is the AppLeS system [26, 27]. Some of the approaches presented in Chapter 7 also provide automatic scheduling support.

  Opus currently requires the user to schedule an application's modules, however automatic scheduling could assist the user in this crucial tasks. An automatic scheduling system for Opus should support both static and dynamic scheduling. Specifically, static scheduling deals with the question of how to derive a good initial SDA distribution at program start which requires only minimal adoption a runtime. This task requires advanced static performance estimators that are able to cope with the underlying HPF and data redistribution systems. An example of such an estimator is P^3T+ [70] which has been developed to work with the HPF compiler VFC as well as the adlib library, both of which are also employed in our Opus system.

  Subsequently, dynamic scheduling is required to adopt a running application. Apart from monitoring the application behavior, dynamic information about the hardware capabilities is required. Systems such as the network weather service [185], JavaSymphony [69], or the Globus components GRAM [63] and MDS [73] may provide the required data.

  Opus provides the **DYNAMIC** attribute to specify SDAs that may be subject to migration. By not associating this attribute the user may enforce an invariant SDA location even in the presence of dynamic scheduling systems.

- *Integration of different HPF systems*
  The implementation of Opus discussed in Chapter 5 relies on an underlying HPF compiler that handles the data parallel portions of Opus. Currently, the HPF compiler VFC [21] is employed. Relying on a single system hampers portability since we are able to port Opus only to systems supported by VFC. In order to enhance the portability of our system we plan to integrate our Opus system with other HPF compilers such as Adaptor [35] or the PGI compiler [179]. Technical issues in integrating HPF compilers others than VFC have been discussed in Chapter 5.

  By integrating different HPF compilation systems with Opus they become interoperable by means of the SDA concept. As a consequence mature HPF compilation technology may be leveraged on any system.

- *Metacomputing issues*
  Opus has specifically been designed for complex clusters. As discussed in Chapter 1 complex clusters reduce the requirements of more general metacomputing platforms. Nevertheless, Opus might also be employed on metacomputers; given support for some crucial issues is provided:

- *Security* is one of the most important issues in metacomputing [147]. In metacomputing systems it is not only necessary to protect users and their applications from malicious interventions but also computing systems from harmful programs. Java, for instance, verifies dynamically down-loaded code prior to execution. Systems such as CorbaServices [151] or Globus [81] include low level security infrastructures whose adoption for our Opus system we will examine.
- Closely related to security is *accounting* which provides means to track, limit, or charge for the consumption of resources in a system [147]. Again, metacomputing systems such as Corba or Globus provide the infrastructure required to set up according policies.
- *Fault tolerance* is finally a crucial issue not only in metacomputing but potentially also (although reduced) in complex clusters. Opus currently does not provide explicit support for fault tolerance, although persistent SDAs may be employed to implement basic checkpoint-restart mechanisms. Prior to devising fault recovery strategies fault detection in distributed systems is required [169]. Given such fault detection services several fault recovery mechanisms, including checkpoint-restart, task migration, etc., can be incorporated. Most of the metacomputing middleware systems discussed in Chapter 7 incorporate some support for fault recovery [97, 148, 161]. We will study these techniques in order to adopt them for our Opus system.

- *Additional language bindings*

  Language interoperability is one of the key requirements of DHPC. The SDA concept of Opus provides language interoperability in that SDAs of any language for which an Opus binding exists are interoperable. In this thesis we have specifically discussed language interoperability among HPF and Java. Since these two languages themselves define interfaces to other languages, in particular C, Opus supports DHPC applications whose modules are implemented in several different languages.

  In order to enhance Opus' support for language interoperability even further, Opus bindings for other languages will be defined. Since Opus is an object based system object oriented languages, most prominently C++, are preferred. In addition, parallel Java dialects such as HPJava [41, 42] and SPAR [182] could be integrated in our OpusJava framework. Some technical details that arise in this context have already been discussed in Chapter 4.

Bibliography

[1] F. Agrawal, A. Sussman, and J. Saltz. An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and Block Structured Applications. *IEEE Trans. on Parallel and Distributed Systems*, 6(7):747–754, July 1995.

[2] S. Ahmed, N. Carriero, and D. Gelernter. A Program Building Tool for Parallel Applications. In *DIMACS Workshop on Specifications of Parallel Algorithms*, Princeton University, May 1994.

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

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, June 1990.

[5] G. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.

[6] E. Armstrong. HotSpot: A new breed of virtual machine. *JavaWorld: IDG's magazine for the Java community*, March 1998. http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot_p.html.

[7] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition.* Addison-Wesley, 1998.

[8] S. Baden and S. Fink. A Programming Methodology for Dual-tier Multicomputers. *IEEE Transactions on Software Engineering*, 26(3):212–226, 2000.

[9] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, and T. Ruhl. Performance Evaluation of the Orca Shared-Object System. *ACM Trans. on Computer Systems*, 16(1):1–40, February 1998.

[10] H. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3):74–84, 1998.

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

[12] H. Bal, J. Steiner, and A. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[13] A. Barak and O. La'adan. The MOSIX Mulitcomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[14] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[15] P. Beckman, P. Fasel, W. Humphrey, and S. Mniszewski. Efficient Coupling of Parallel Applications Using PAWS. In *Proc. HPDC*, Chicago, IL, July 1998.

[16] P. Beckman and D. Gannon. Tulip: A Portable Run-Time System for Object-Parallel Systems. In *Proc. IPPS'96*, Honolulu, Hawaii, 1996.

[17] 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.

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

[19] S. Ben Hassen, H. Bal, and C. Jacobs. A task- and data-parallel programming language based on shared objects. *ACM Transactions on Programming Languages and Systems*, 20(6):1131–1170, November 1998.

[20] R. Ben-Natan. *Objects on the Web - Designing, Building, and Deploying Object-Oriented Applications for the Web.* McGraw-Hill, 1997.

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

[22] S. Benkner. HPF+: High Performance Fortran for Advanced Scientific and Engineering Applications. *Journal of Future Generation Computer Systems*, 15(3):381–391, 1999.

[23] S. Benkner. Optimizing Irregular HPF Applications Using Halos. In *Workshop Proceedings of the International Symposium on Parallel Processing*, San Juan, Puerto Rico, April 1999.

[24] S. Benkner, E. Laure, and H. Zima. HPF+: An Extension of HPF for Advanced Industrial Applications. Technical Report TR99-01, Institute for Software Technology and Parallel Systems, University of Vienna, 1999.

[25] S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima. High-Level Management of Communication Schedules in HPF-like Languages. In *Proc. Int. Conf. on Supercomputing (ICS'98)*, Melbourne, Australia, July 1998.

[26] F. Berman. High-Performance Schedulers. In I. Foster and C. Kesselman, editors, *The GRID. Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan Kaufmann, 1999.

[27] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Supercomputing'96*, Pittsburgh, PA, November 1996.

[28] P. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, February 1996.

[29] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[30] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. 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.

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

[32] Blue Gene Architecture. http://www.research.ibm.com/news/detail/architecture_fact.html.

[33] T. Brandes. Exploiting Advanced Task Parallelism in High Performance Fortran via a Task Library. In *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685. Springer-Verlag, 1999.

[34] T. Brandes and S. Benkner. Exploiting Data Locality on Scalable Shared Memory Machines with Data Parallel Programs. In *EuroPar 2000 Parallel Processing*, Lecture Notes in Computer Science, No.1900. Springer-Verlag, 2000.

[35] T. Brandes and F. Zimmermann. ADAPTOR—A Transformation Tool for HPF Programs. In *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser Verlag, April 1994.

[36] F. Breg, S. Diwan, J. Villacis, et al. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11-13):941–955, 1998.

[37] R. Campbell. *Path Expressions: A technique for specifying process synchronization.* PhD thesis, Computing Laboratory, The University of Newcastle Upon Tyne, 1976.

[38] R. Campbell and A. Habermann. The Specification of Process Synchronization by Path Expressions. In G. Goos and J. Hartmanis, editors, *LNCS*, volume 16. Springer Verlag, 1974.

[39] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11-13):1043–1061, 1998.

[40] B. Carpenter, Y.-J. Chang, G. Fox, and X. Li. Java as a Language for Scientific Parallel Programming. In *10th International Workshop on Languages and Compilers for Parallel Computing*, LNCS, No. 1366. Springer, 1998.

[41] B. Carpenter, G. Zhang, G. Fox, et al. Towards a Java Environment for SPMD Programming. In *Proceedings EuroPar '98*, LNCS, No. 1470, Southampton, 1998. Springer.

[42] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. *HPJava*: data parallel extensions to Java. *Concurrency: Practice and Experience*, 10(11-13):873–877, 1998.

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

[44] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1):40–49, January 1995.

[45] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.

[46] N. Carriero and D. Gelernter. Data Parallelism and Linda. In *Languages and Compilers for Parallel Computing, 5th International Workshop*, Lecture Notes in Computer Science, No. 757. Springer, 1992.

[47] 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):212–223, Fall 1997.

[48] H. Casanova and J. Dongarra. The use of Java in the NetSolve project. In *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, Berlin, 1997.

[49] H. Casanova and J. Dongarra. NetSolve Version 1.2: Design and Implementation. Technical Report UT-CS-98-406, Department of Computer Science, University of Tennessee, November 1998.

[50] H. Casanova and J. Dongarra. Using Agent-based Software for Scientific Computing in the NetSolve System. *Parallel Computing*, 24(12-13):1777–1790, November 1998.

[51] M. Chandy, I. Foster, K. Kennedy, et al. Integrated Support for Task and Data Parallelism. *International Journal of Supercomputer Applications*, 8(2):80–98, Summer 1994.

[52] M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. In *Languages and Compilers for Parallel Computing, 5th International Workshop*, Lecture Notes in Computer Science, No. 757, pages 124–144. Springer Verlag, August 1992.

[53] 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. MIT Press, 1993.

[54] 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.

[55] 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.

[56] B. Chapman and P. Mehrotra. OpenMP and HPF: Integrating Two Paradigms. In *Proceedings EuroPar'98*, LNCS, No. 1470. Springer, September 1998.

[57] B. Chapman, P. Mehrotra, and H. Zima. Enhancing OpenMP With Features for Locality Control. In *Proc. ECMWF Workshop "Towards Teracomputing – The Use of Parallel Processors in Meteorology"*, Reading, England, November 1998.

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

[59] S. Chatterjee, J. Gilbert, F. 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.

[60] B. Christiansen, P. Cappello, M. Ionescu, M. Neary, K. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, November 1997.

[61] D. Cronk, M. Haines, and P. Mehrotra. Thread Migration in the Presence of Pointers. In *Proc. 30^{th} Hawaii International Conference on System Sciences*, January 1997.

[62] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufman, 1999.

[63] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[64] V. Decyk, C. Norton, and B. Szymanski. Expressing Object-Oriented Concepts in Fortran90. *ACM Fortran Forum*, 16(1), April 1997.

[65] M. Dempster and R. Thompson. Parallelization and Aggregation of Nested Benders Decomposition. In *Proceedings of the APMOD95 Conference*, Annals of Operations Research 81, 1995.

[66] 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.

[67] Earth Simulator Research and Development Center. Outline of the Earth Simulator. http://www.gaia.jaeri.go.jp/.

[68] G. Fagg, 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*, LNCS, No. 1332. Springer, 1997.

[69] T. Fahringer. JavaSymphony: A System for Development of Locality-Oriented Distributed and Parallel Java Applications. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000)*, Chemnitz, Germany, November 2000. IEEE Computer Society.

[70] T. Fahringer and A. Pozgaj. P^3T+: A Performance Estimator for Distributed and Parallel Programs. *Journal of Scientifc Programming*, 7(1), November 2000.

[71] S. Fink, S. Baden, and S. Kohn. Flexible Communication Mechanisms for Dynamic Structured Applications. In *Proc. 3rd Int'l Workshop IRREGULAR '96*, August 1996.

[72] S. Fink, S. Baden, and S. Kohn. Efficient Run-time Support for Block-Structured Applications. *Journal of Parallel and Distributed Computing*, 50(1):61–82, 1998.

[73] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, and W. Smith. A Directory Service for Configuring High-Performance Distributed Computations. In *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*, 1997.

[74] I. Foster, B. Avalani, M. Xu, and A. Choudhary. A Compilation System That Integrates High Performance Fortran and Fortran M. In *Proceedings 1994 Scalable High Performance Computing Conference*. IEEE Computer Science Press, 1994.

[75] I. Foster and M. Chandy. Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[76] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. Wide-Area Implementation of the Message Passing Interface. *Parallel Computing*, 24(12):1735–1749, 1998.

[77] 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):35–48, Jan. 1997.

[78] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl. J. Supercomputer Applications*, 11(2):115–128, 1997.

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

[80] I. Foster and C. Kesselman, editors. *The Grid*. Morgan Kaufmann, 1999.

[81] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

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

[83] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel Computing*, 37(1):70–82, August 1996.

[84] I. Foster, D. 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.

[85] G. Fox and W. Furmanski. High Performance Commodity Computing. In I. Foster and C. Kesselman, editors, *The GRID. Blueprint for a New Computing Infrastructure*, pages 237–255. Morgan Kaufmann, 1999.

[86] E. Gallopoulos, E. Houstis, and J. Rice. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science & Engineering*, 1(2):11–23, Summer 1994.

[87] D. Gannon et al. Developing Component Architectures for Distributed Scientific Problem Solving. *IEEE Computational Science & Engineering*, 5(2):50–63, April-June 1998.

[88] D. Gannon et al. HPC++ and the HPC++Lib Toolkit. In *Compilation Issues on Distributed Memory Systems*. Springer Verlag, 1998.

[89] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[90] M. Gokhale, B. Holmes, and K. Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer*, 28(4):23–31, April 1995.

[91] Grid Computing Environments Working Group. http://www.computingportals.org.

[92] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An Operating System for Wide-Area Computing. Technical Report CS-99-12, Department of Computer Science, University of Virginia, March 1999.

[93] A. Grimshaw, W. Wulf, et al. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, January 1997.

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

[95] J. Grosch and H. Engelmann. A Tool Box for Compiler Construction. In *LNCS, No. 447*, pages 106–116. Springer Verlag, October 1990.

[96] 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.

[97] Object Management Group. Fault Tolerant CORBA. OMG TC Document orbos/99-12-08, December 1999.

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

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

[100] 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.

[101] 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.

[102] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[103] G. Hamilton (Editor). *JavaBeans V 1.01.* SUN Microsystems, 1997.

[104] T. Haupt, E. Akarsu, and G. Fox. WebFlow: A Framework for Web Based Metacomputing. In *Proceedings HPCN Europe 1999*, LNCS, No. 1593. Springer, 1999.

[105] T. Haupt, E. Akarsu, G. Fox, and W. Furmanski. Web based metacomputing. *Future Generation Computer Systems*, 15(5-6):735–743, 1999.

[106] High Performance Fortran Language Specification Version 1.1, November 1994.

[107] High Performance Fortran Language Specification Version 2.0, January 1997.

[108] S. Hiranandani, K. Kennedy, et al. An Overview of the Fortran D Programming System. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[109] C. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[110] M. Horstmann and M. Kirtland. DCOM Architecture.
MSDN Online Library, http://msdn.microsoft.com/library/, July 1997.

[111] E. Houstis, E. Gallopoulos, R. Bramley, and J. Rice. Problem-Solving Environments for Computational Science. *IEEE Computational Science & Engineering*, 4(3):18–21, July/September 1997.

[112] S. Hummel, T. Ngo, and H. Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, 9(6):621–632, June 1997.

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

[114] IEEE/ANSI. POSIX Threads Extensions. IEEE/ANSI 1003.1c-1995.

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

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

[117] ISO. Fortran 95 Standard. ISO/IEC 1539 :1997.

[118] ISO Working Group WG5. Working Draft J3/01-007.
http://www.j3-fortran.org/Fortran-200x.html.

[119] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *ACM 1999 Java Grande Conference*, San Francisco, California, June 1999.

[120] Java Grande Forum. http://www.javagrande.org.

[121] E. Johnson, D. Gannon, and P. Beckman. HPC++: Experiments with the Parallel Standard Template Library. In *Proc. ICS'97*, Vienna, Austria, 1997.

[122] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[123] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing, Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, California, 1994.

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

[125] P. Launay and J.-L. Pazat. A Framework for Parallel Programming in Java. In *Proceedings HPCN Europe 1998*, Lecture Notes in Computer Science, No. 1401. Springer, April 1998.

[126] E. Laure. ParBlocks - A New Methodology for Specifying Concurrent Method Executions in Opus. In P. Amestoy et al., editors, *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685. Springer-Verlag, 1999.

[127] E. Laure. ParBlocks - A New Methodology for Specifying Concurrent Method Executions in Opus. Technical Report TR99-05, Institute for Software Technology and Parallel Systems, University of Vienna, 1999.

[128] E. Laure. Distributed High Performance Computing with OpusJava. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Parallel Computing: Fundamentals & Applications, Proceedings of the International Conference ParCo'99, 17-20 August 1999, Delft, The Netherlands*, pages 590–597. Imperial College Press, April 2000.

[129] E. Laure. OpusJava: A Java Framework for Distributed High Performance Computing. *Future Generation Computer Systems*, in print 2001.

[130] E. Laure, M. Haines, P. Mehrotra, and H. Zima. Compiling Data Parallel Tasks for Coordinated Execution. In P. Amestoy et al., editors, *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685. Springer-Verlag, 1999.

[131] E. Laure, M. Haines, P. Mehrotra, and H. Zima. On the Implementation of the Opus Coordination Language. *Concurrency: Practice and Experience*, 12(4):227–249, April 2000.

[132] E. Laure, E. Mehofer, H. Moritsch, V. Sipkova, and A. Swietanowski. HPF in Financial Management Under Uncertainty. Technical Report 1999-21, SFB Aurora, October 1999.

[133] E. Laure, P. Mehrotra, and H. Zima. Opus: Heterogeneous Computing With Data Parallel Tasks. *Parallel Processing Letters*, 9(2):275–289, June 1999.

[134] M. Leair, J. Merlin, S. Nakamoto, V. Schuster, and M. Wolfe. Distributed OMP - A Programming Model For SMP Clusters. In *Eight International Workshop on Compilers for Parallel Computers*, pages 229–238, Aussois, France, January 2000.

[135] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[136] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Proceedings of the seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, USA, May 1999. ACM.

[137] P. Mehrotra, J. Van Rosendale, and H. Zima. High Performance Fortran: History, Status and Future. *Parallel Computing*, 24(3-4):325–354, 1998.

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

[139] J. Merlin, S. Baden, S. Fink, and B. Chapman. Multiple Data Parallelism with HPF and KeLP. *Future Generation Computing Systems*, 15(3):393–405, May 1999.

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

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

[142] P. Messina. Distributed Supercomputing Applications. In I. Foster and C. Kesselman, editors, *The GRID. Blueprint for a New Computing Infrastructure*, pages 55–73. Morgan Kaufmann, 1999.

[143] B. Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice Hall, 1997.

[144] J. Moreira, S. Midkiff, and M. Gupta. From Flop to Megaflops: Java for Technical Computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, March 2000.

[145] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, M. Snir, and R. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.

[146] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *ACM 1999 Java Grande Conference*, San Francisco, June 1999.

[147] C. Neuman. Security, Accounting, and Assurance. In I. Foster and C. Kesselman, editors, *The GRID. Blueprint for a New Computing Infrastructure*, pages 396–420. Morgan Kaufmann, 1999.

[148] A. Nguyen-Tuong and A. Grimshaw. Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications. *Parallel Processing Letters*, 9(2):291–301, June 1999.

[149] R. Numrich and J. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[150] Object Management Group. *Common Facilities Architecture 4.0*, November 1995.

[151] Object Management Group. *CORBAservics: Common Object Services Specification*, March 1995.

[152] Object Management Group. *CORBAtelecoms: Telecommunications Domain Specifications 1.0*, June 1998.

[153] Object Management Group. *CORBAmanufacturing: Manufacturing Domain Specifications 1.0*, October 1999.

[154] Object Management Group. *CORBAMed: Healthcare Domain Specifications 1.0*, March 1999.

[155] Object Management Group. *The Common Object Request Broker: Architecture and Specification 2.3*, June 1999.

[156] *OpenMP C and C++ Application Program Interface Version 1.0.* http://www.openmp.org/, October 1998.

[157] S. Orlando and R. Perego. $COLT_{HPF}$, a run-time support for the high-level co-ordination of HPF tasks. *Concurrency: Practice and Experience*, 11(8):407–434, 1999.

[158] S. Parker, C. Johnson, and D. Beazley. Computational Steering Software Systems and Strategies. *IEEE Computational Science & Engineering*, 4(4):50–59, October/December 1997.

[159] M. Philippsen and B. Haumacher. More Efficient Object Serialization. In *Proc. International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, April 1999.

[160] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[161] J. Plank, H. Casanova, M. Beck, and J. Dongarra. Deploying Fault-Tolerance and Task Migration with NetSolve. In *Lecture Notes in Computer Science, No. 1541*, 1998.

[162] 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.

[163] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *Proceedings IEEE Aerospace Conference*, 1997.

[164] M. Rinard, D. Scales, and M. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6):28–38, June 1993.

[165] T. Rühl, H. Bal, R. Bhoedjang, K. Langendoen, and G. Benson. Experience with a Portability Layer for Implementing Parallel Programming Systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, Sunnyvale, California, August 1996.

[166] D. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[167] L. Smarr and C.E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):45–52, June 1992.

[168] J. Squyres, A. Lumsdaine, W. George, J. Hagedorn, and J. Devaney. The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI. In *Proceedings, MPIDC'2000*, March 2000.

[169] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A Fault Detection Service for Wide Area Distributed Computations. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, 1998.

[170] T. Sterling and L. Bergman. A Design Analysis of a Hybrid Technology Multithreaded Architecture for Petaflops Scale Computation. In *International Conference on Supercomputing*. ACM, June 1999. see also: http://htmt.cacr.caltech.edu/Overview.html.

[171] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, third edition, 1997.

[172] SUN Microsystems. *XDR: External Data Representation Standard, RFC 1014*. Network Working Group, June 1987.

[173] SUN Microsystems. *Java Native Interface Specification*, 1997.

[174] SUN Microsystems. *Java Object Serialization Specification*, 1998.

[175] SUN Microsystems. *Java Remote Method Invocation Specification*, 1998.

[176] SUN Microsystems. *JavaSpaces Specification*, 1999.

[177] SUN Microsystems. *Sun MPI 4.0 Programming and Reference Guide*, 1999.

[178] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[179] The Portland Group, Inc. *PGHPF User's Guide*, 2000.

[180] TOP500 Supercomputer Sites. http://www.top500.org/.

[181] C. van Reeuwijk. *Vnus language specification Version 2.1 beta 1.* Delft University of Technology, 1999.

[182] C. van Reeuwijk, A. van Gemund, and H. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, November 1997.

[183] M. van Steen, P. Homburg, and A.S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, January-March 1999.

[184] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications*, 35(2):46–55, February 1997.

[185] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6), 1999.

[186] Z. Yang and K. Duddy. CORBA: A Platform for Distributed Object Computing. *ACM Operating Systems Review*, 30(2):4–31, April 1996.

Appendix A

Opus Syntax

A.1 Syntax Metalanguage

The notation used in the syntax rules is a variation of Backus-Naur form and in accordance with the Fortran 90 standard document [116].

The syntactic metasymbols used are:

is	introduces a syntactic class definition
or	introduces a syntactic class alternative
[]	encloses an optional item
[]...	encloses an optionally repeated item which may occur zero or more times

Assumed syntax rules are:

xyz-list	**is**	*xyz* [, *xyz*]...
xyz-name	**is**	*name*(R304)
integer-xyz	**is**	*xyz*
xyz-variable	**is**	*variable*(R601)

Nonterminal symbols which are part of Fortran 90 are implicitly defined according to the Fortran 90 language specification [116] and annotated with the Fortran 90 rule number (prefix R). The following nonterminals are taken from the Fortran 90 standard document:

actual-arg-spec
attr-spec
char-literal-constant
contains-stmt
dummy-arg
end-function-stmt
end-subroutine-stmt
entity-decl
execution-part
expr
function-reference
function-stmt
integer-constant
interface-body
name
scalar-logical-expr
section-subscript
specification-expr
specification-part
subroutine-stmt
variable

A **constraint** is a restriction on the syntax rules that is capable of being checked by the compiler.

A.2 SDA Type

sda-type	**is**	*sda-stmt* [*sda-specification-part*] [*sda-method-part*] [*parblock-directive*]... *end-sda-stmt*
sda-stmt	**is**	`SDA TYPE` [[`,` *sda-attribute*] `::`] *type-name* [`(` *dummy-arg-list*`(R1221)` `)`]
sda-attribute	**is**	`LIGHTWEIGHT`
sda-specification-part	**is**	*specification-part*`(R204)`
sda-method-part	**is**	*contains-stmt*`(R1225)` *method* [*method*]...
method	**is** **or**	*function-method* *subroutine-method*
function-method	**is**	*function-stmt*`(R1216)` [*condition-clause*] *subprogram-body* *end-function-stmt*`(R1218)`
condition-clause	**is**	`WHEN (` *scalar-logical-expr*`(R725)` `)`
subprogram-body	**is**	[*specification-part*`(R204)`] [*execution-part*`(R208)`]
subroutine-method	**is**	*subroutine-stmt*`(R1220)` [*condition-clause*] *subprogram-body* *end-subroutine-stmt*`(R1222)`
end-sda-stmt	**is**	`END` [`SDA TYPE` [*type-name*]]

A.3 SDA Declaration

sda-declaration-stmt	**is**	*sda-type-spec* [[`,` *attr-spec-list*`(R503)`] [`,` *sda-attr-spec*] `::`] *entity-decl-list*`(R504)`
sda-type-spec	**is**	`SDA` [`(` *sda-type-name* `)`]
sda-attr-spec	**is**	`DYNAMIC`

Constraint: `POINTER` attribute is not allowed in *attr-spec-list*.

A.4 Method Activation

method-activation	**is**	*s-task-activation*
	or	*a-task-activation*
	or	*function-method-ref*
s-task-activation	**is**	[`CALL`] *method-ref*
method-ref	**is**	*sda-variable* . *procedure-ref*
		[, *status-clause*][, *priority-clause*]
status-clause	**is**	`STAT =` *stat-variable*
procedure-ref	**is**	*subroutine-name*
		[([*actual-arg-spec-list*(`R1211`)])]
	or	*intrinsic-method-call*
a-task-activation	**is**	[*event-clause*] `SPAWN` *task-ref*
task-ref	**is**	*method-ref*
	or	*subroutine-name* [([*actual-arg-spec-list*(`R1211`)])]
		[*creation-attr*][, *status-clause*][, *priority-clause*]
event-clause	**is**	*event-variable* =
function-method-ref	**is**	*sda-variable* . *function-reference*(`R1209`)

A.5 Intrinsic Methods

intrinsic-method-call	**is**	*sda-creation*
	or	*sda-termination*
	or	*sda-storage*
	or	*sda-access*
	or	*sda-migration*
sda-creation	**is**	*create-stmt*
	or	*load-stmt*
create-stmt	**is**	`CREATE` [(*actual-arg-spec-list*(`R1211`))]
		[*creation-attr*]
creation-attr	**is**	`ON (` *resource-request* `)`
load-stmt	**is**	`LOAD (` *external-sda-identification* `)`
		[*creation-attr*]
external-sda-identification	**is**	*char-variable*
	or	*char-literal-constant*(`R420`)
sda-termination	**is**	`TERMINATE`
	or	`KILL`
sda-storage	**is**	`SAVE (` *external-sda-identification* `)`
sda-access	**is**	`GET(` *sda-component* , *variable*(`R601`) `)`
	or	`PUT(` *sda-component* , *expr*(`R723`) `)`
sda-component	**is**	*variable*(`R601`)

sda-migration **is** `MIGRATE` *creation-attr*

resource-request **is** [*machine-request*,] [*processor-request*]

machine-request **is** `MACHINE =` *machine-spec*

machine-spec **is** *direct-mspec*
or *indirect-spec*

direct-mspec **is** *machine-name*

indirect-spec **is** *sda-variable*
or *task-specification*
or `SELF`

task-specification **is** *event-variable*

processor-request **is** `PROCESSORS =` *processor-spec*

processor-spec **is** [*allocation-strategy*] *specification-expr*`(734)`
or *processors-name* [`(` *section-subscript-list*`(R618) )`]

allocation-strategy **is** `NEW`
or `SAME`

Constraint: The creation of lightweight SDAs must not contain a *machine-request*.

Constraint: A *resource-request* that does not contain a *machine-request* must contain a *processor-request*.

Constraint: The `NEW` allocation strategy must not be applied to lightweight SDAs.

Constraint: In the *section-subscript-list*, the number of *section-subscripts* must equal the rank of the *processor-name*.

A.6 Task Synchronization

wait-stmt **is** `WAIT(` *event-variable* `)` [`,` *status-clause*]

A.7 Intra-SDA Parallelism

parblock-directive **is** *oc-directive-origin parblock-stmt parblock-body-list*

oc-directive-origin **is** `!OC$`

parblock-stmt **is** `PARBLOCK`

parblock-body **is** `(` *par-section* `)`

par-section **is** *method-name-list*

A.8 Priority Scheduling

priority-clause	**is**	`PRIORITY =` *priority-value*
priority-value	**is**	*integer-variable*
	or	*integer-constant*`(R308)`
	or	`MAX-PRIORITY`
	or	`MIN-PRIORITY`

A.9 SDA Interface

sda-interface	**is**	*sda-interface-stmt*
		interface-body`(R1204)`
		end-sda-interface-stmt
sda-interface-stmt	**is**	`SDA INTERFACE` *sda-name*
end-sda-interface-stmt	**is**	`END` [`SDA INTERFACE` [*sda-name*]]

Appendix B

Intrinsic Functions

In the following we summarize the intrinsic functions provided by Opus. We present here only those intrinsic functions which are not in HPF or Fortran 90, respectively.

B.1 Task Inquiry and Synchronization Functions

B.1.1 WAITANY(E)

Description. Blocks the calling task until any task associated with `E` has finished its execution.

Class. Private task synchronization function.

Argument. `E` must be a rank one array of type event.

Result Type. Integer.

Result Value. The index within array `E` of the event a finished task is associated with.

B.1.2 TEST(E)

Description. Determines the execution status of the task associated with `E`.

Class. Private task inquiry function.

Argument. `E` must be a scalar of type event.

Result Type. Logical.

Result Value. The result is true if the task associated with `E` has finished its execution.

B.2 SDA Inquiry Functions

B.2.1 CREATED(S)

Description. Determines whether `S` is associated with an SDA object.

Class. Public SDA inquiry function.

Argument. `S` must be a scalar of type SDA.

Result Type. LOGICAL.

Result Value. The result is true if `S` is associated with an SDA object.

B.2.2 EQUAL(S)

Description. Determines whether two SDA variables are associated with the same SDA object.

Class. Public SDA inquiry function.

Argument. `S` must be a scalar of type SDA.

Result Type. Logical.

Result Value. The result is true if `S` is associated with an SDA object and this SDA object is the same one as the SDA object to which the function is applied.

Appendix C

OpusJava API

In the following, the API for OpusJava, in particular for the classes `SDA`, `Request`, `Resource`, and `Event`, is specified. Note that all the necessary exceptions are omitted for sake of conciseness. In presenting the API we follow closely Java's API style.

C.1 Class SDA

```
java.lang.Object
  |
  +--SDA
```

```
public final class SDA
extends Object
implements Serializable
```

The SDA class is a stub for SDA objects. It establishes a permanent RMI connection to the SDA object (which is either newly instantiated or loaded from persistent memory) and provides methods for invoking methods of the SDA in a synchronous or asynchronous manner. Support for terminating and migrating SDAs is provided, too. All input and output arguments to SDA methods have to implement the `Serializable` interface.

Constructors

`public SDA(String name)`

Constructs a new SDA object and contacts the `SDAServer` of the local machine in order to create a new object of class `name` as an SDA.

Parameters:

`name` - the class name of the object to be created as SDA.

`public SDA(String filename, String codebase)`

Constructs a new SDA object and contacts the `SDAServer` of the local machine in order to load a persistent SDA from a file.

Parameters:

`filename` - the file name of the persistent SDA.

`codebase` - the codebase where the persistent SDA is stored.

`public SDA(String name, Object[] args)`

Constructs a new SDA object and contacts the `SDAServer` of the local machine in order to create a new object of class `name` as an SDA, passing `args` as input arguments for the constructor of the new SDA object.

Parameters:

`name` - the class name of the object to be created as SDA.

`args` - the input arguments for the constructor of class `name`.

`public SDA(String name, Resource machine)`

Constructs a new SDA object and contacts the `SDAServer` of `machine` in order to create a new object of class `name` as an SDA. The number of processors to be used is embodied in the `Resource` object.

Parameters:

`name` - the class name of the object to be created as SDA.

`machine` - The resource object containing the name of the machine on which the SDA should be created and the number of processors to be used.

`public SDA(String filename, String codebase, Resource machine)`

Constructs a new SDA object and contacts the `SDAServer` of `machine` in order to load a persistent SDA from a file. The number of processors to be used is embodied in the `Resource` object.

Parameters:

`filename` - file name of the persistent SDA.

`codebase` - the codebase where the persistent SDA is stored.

`machine` - The resource object containing the name of the machine on which the SDA should be created and the number of processors to be used.

`public` **SDA**`(String name, Object[] args, Resource machine)`

Constructs a new SDA object and contacts the **`SDAServer`** of **`machine`** in order to create a new object of class **`name`** as an SDA, passing **`args`** as input arguments for the constructor of the new SDA object.

Parameters:

`name` - the class name of the object to be created as SDA.

`args` - the input arguments for the constructor of class **`name`**.

`machine` - The resource object containing the name of the machine on which the SDA should be created and the number of processors to be used.

`public` **SDA**`(Request r)`

Constructs a new SDA object and contacts the **`SDAServer`** of the local machine in order to create a new object as described in **`Request`**.

Parameters:

`r` - The **`Request`** object containing the name of the class that is to be created as an SDA together with the necessary input arguments for the constructor.

`public` **SDA**`(Request r, Resource machine)`

Constructs a new SDA object and contacts the **`SDAServer`** of **`machine`** in order to create a new object as described in **`Request`**.

Parameters:

`r` - The **`Request`** object containing the name of the class that is to be created as an SDA together with the necessary input arguments for the constructor.

`machine` - The resource object containing the name of the machine on which the SDA should be created and the number of processors to be used.

Methods

`public Object` **call**`(String method)`

Invokes **`method`** in the SDA in a synchronous manner. Results are returned in **`Object`**.

Parameters:

`method` - the name of the method to be invoked.

Returns:

The results encoded in an **`Object`**.

`public Object` **call**`(String method, Object args[])`

Invokes **`method`** in the SDA in a synchronous manner passing **`args`** as input arguments. Results are returned in **`Object`**.

Parameters:

`method` - the name of the method to be invoked.

`args` - an array of input arguments.

Returns:

The results encoded in an **`Object`**.

`public Object` **call**`(Request r)`

Invokes the method described in `r` within the SDA in a synchronous manner. Results are returned in `Object`.

Parameters:

`r` - The `Request` object containing the name of the method that is to be invoked together with the necessary input arguments.

Returns:

The results encoded in an `Object`.

`public Event` **spawn**`(String method)`

Invokes `method` in the SDA in an asynchronous manner. An `Event` is returned immediately, which acts as a reference to the call and contains also the results of the method.

Parameters:

`method` - the name of the method to be invoked.

Returns:

A new `Event` object bound to the request.

`public Event` **spawn**`(String method, Object args[])`

Invokes `method` in the SDA in an asynchronous manner passing `args` as input arguments. An `Event` is returned immediately, which acts as a reference to the call and contains also the results of the method.

Parameters:

`method` - the name of the method to be invoked.

`args` - an array of input arguments.

Returns:

A new `Event` object bound to the request.

`public Event` **spawn**`(Request r)`

Invokes the method described in `r` within the SDA in an asynchronous manner. An `Event` is returned immediately, which acts as a reference to the call and contains also the results of the method.

Parameters:

`r` - The `Request` object containing the name of the method that is to be invoked together with the necessary input arguments.

Returns:

A new `Event` object bound to the request.

`public boolean` **equals**`(Object obj)`

Compares this object to the specified object `obj`. The result is `true` if and only if the argument is not `null` and is an `SDA` object that refers to the same object as this object.

Parameters:

`obj` - the object to compare with.

Returns:

`true` if the objects are the same; `false` otherwise.

`public void` **`terminate`**`()`

Terminates the remote SDA. In particular native SDAs are terminated by invoking their termination routines which free the memory in use. The RMI link is disconnected.

`public void` **`migrate`**`(Resource machine)`

An existing SDA object is migrated to new resources **`machine`**. An object subject to migration has to implement the **`Serializable`** interface.

Parameters:

`machine` - The resource object containing the name of the machine to which the SDA should be migrated and the number of processors to be used.

`public void` **`save`**`(String filename, String codebase)`

An existing SDA object is stored to persistent memory. This object has to implement the **`Serializable`** interface.

Parameters:

`filename` - file name of the persistent SDA.

`codebase` - the codebase where the persistent SDA is to be stored.

C.2 Class Request

```
java.lang.Object
  |
  +--Request
```

---

public class **Request**
extends Object
implements Serializable

The Request class represents an OpusJava request, either for SDA creation or for method invocation, and encapsulates the name of the SDA (the method) and the required input arguments. A request object may not be altered after its creation. Therefore, only constructors and no public methods or fields are provided.

Constructors

public **Request**(String name)

Constructs a new Request object and initializes it with the name of the SDA (Method) to which the request applies.
Parameters:
`name` - the name of the SDA that is to be created resp. the name of the method that is to be invoked.

public **Request**(String name, Object[] args)

Constructs a new Request object and initializes it with the name of the SDA (Method) to which the request applies together with an array of input arguments.
Parameters:
`name` - The name of the SDA that is to be created resp. the name of the method that is to be invoked.
`args` - The input argument to the request.

public **Request**(String name, Class c)

Constructs a new Request object and initializes it with the name of the SDA (Method) to which the request applies together with a class for storing the results of the method invocation.
Parameters:
`name` - The name of the SDA that is to be created resp. the name of the method that is to be invoked.
`c` - The class used to store results.

`public` **Request**`(String name, Object[] args, Class c)`

Constructs a new Request object and initializes it with the name of the SDA (Method) to which the request applies together with an array of input arguments and a class for storing the results of the method invocation.

Parameters:

`name` - The name of the SDA that is to be created resp. the name of the method that is to be invoked.

`args` - The input argument to the request.

`c` - The class used to store results.

C.3 Class Resource

```
java.lang.Object
  |
  +--Resource
```

`public final class` **Resource**
`extends Object`
`implements Serializable`

The Resource class represents the resources of an SDA. It contains the name of the machine as well as the number of processors available to an SDA. A resource object may not be altered after its creation. Therefore, only constructors and no public methods or fields are provided.

Constructors

`public` **Resource**`(String machine, int processors)`

Constructs a new Resource object and initializes it with the name of the associated **`machine`** and the number of **`processors`**.

Parameters:

`machine` - the name (typically an IP-address) of the machine associated with the resource object.

`processors` - the number of processors available to the resource object on the associated machine.

C.4 Class Event

```
java.lang.Object
  |
  +--Event
```

---

public final class Event
extends Object
implements Serializable

The Event class represents a link to asynchronously spawned methods of an SDA. It is typically only created as return value of `spawn` methods. Events allow queries about the status of an asynchronous method invocation and encode the results of such methods, as well. It provides methods for testing a method invocation's status in a non-blocking fashion, for waiting on the return, and for accessing the results.

Methods

`public boolean` **testEvent**`()`

Tests the status of a method invocation. If the method execution has already terminated, `true` is returned, otherwise the result is `false`.

Returns:
The execution status of a remote method invocation.

`public void` **waitEvent**`()`

Blocks the caller until the remote method referenced has finished its execution.

`public Object` **getResults**`()`

The results of an asynchronous method execution are returned. In the case the execution has not yet finished, the caller is blocked until the results are available.

Returns:
The results of the associated method encoded within an Object.
