An Overview of the Opus Language and Runtime System*

Piyush Mehrotra **Matthew Haines**

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

Abstract

We have recently introduced a new language, called *Opus*, which provides a set of Fortran language extensions that allow for integrated support of task and data parallelism. It also provides shared data abstractions (SDAs) as a method for communication and synchronization among these tasks. In this paper, we first provide a brief description of the language features and then focus on both the language-dependent and language-independent parts of the runtime system that support the language. The language-independent portion of the runtime system supports lightweight threads across multiple address spaces, and is built upon existing lightweight thread and communication systems. The language-dependent portion of the runtime system supports conditional invocation of SDA methods and distributed SDA argument handling.

1 Introduction

Data parallel language extensions, such as High Performance Fortran (HPF) [19] and Vienna Fortran [3], are adequate for expressing and exploiting the data parallelism in scientific codes. However, there are a large number of scientific and engineering codes which exhibit multiple levels of parallelism. *Multidisciplinary optimization* (MDO) applications are a good example of such codes. These applications, such as weather modeling and aircraft design, integrate codes from different disciplines in a complex system in which the different discipline codes can execute concurrently, interacting with each other only when they need to share data. In addition to this outer level of task parallelism, the individual discipline codes often exhibit internal data parallelism. It is desirable to have a single language which can exploit both task and data parallelism, providing control over the communication and synchronization of the coarse-grain tasks.

We have recently designed a new language, called *Opus*, which extends HPF to provide the support necessary for coordinating the parallel execution, communication, and synchronization of

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

such codes [4]. Along with extensions to manage independently executing tasks, we have introduced a new mechanism, called *Shared Data Abstractions* (SDAs), which allows these tasks to share data with each other. SDAs generalize Fortran 90 modules by including features from both *objects* in object-oriented languages and *monitors* in shared memory languages. The result is a data structure mechanism that provides a high-level, controlled interface for large grained parallel tasks to interact with each other in a uniform manner.

In a previous paper [4], we introduced the Opus language syntax and semantics for specifying task parallelism and their interaction with SDAs. In this paper, focus our attention on the runtime system that supports these mechanisms. In particular, we describe both the language-dependent and language-independent portions of the runtime system. The language-independent portion of the runtime system is based on our lightweight threads package called *Chant* [17]. This provides an interface for lightweight, user-level threads which have the capability of communication and synchronization across separate address spaces. Chant extends lightweight thread and communication libraries to provide a simple, coherent, and efficient interface for using lightweight threads in a distributed memory environment. Additionally, Chant supports a grouping of threads, called a *rope*, which provides an indexed naming scheme for the set of global threads, as well as providing a scope for global communication, synchronization, and reduction operations. Ropes can be used to implement data parallelism. The language-dependent portion of the runtime system supports conditional invocation of SDA methods and distributed SDA argument handling.

The remainder of the paper is organized as follows: Section 2 briefly outlines the Opus language; Section 3 outlines the runtime support necessary for supporting the task and shared data abstraction extensions in Opus; and Section 4 discusses related research.

2 The Opus Language

In Opus, a program is composed of a set of asynchronous, autonomous tasks that execute independently of one another. These tasks may embody nested parallelism, for example, by executing a data parallel HPF program. A set of tasks interact by creating an SDA object of an appropriate type and making the object accessible to all tasks in the set. The SDA executes autonomously on its own resources, and acts as a data repository. The tasks can access the data within an SDA object by invoking the associated SDA public methods, which execute asynchronously with respect to the invoking task. However, the SDA semantics enforce exclusive access to the data for each call to the SDA. This is done by ensuring that only one method of a particular SDA is active at any given time. This combination of task and SDA concepts forms a powerful tool for hierarchically structuring a complex body of parallel code.

We presume that High Performance Fortran (HPF) [19] is to be used to specify the data parallelism in the codes. Thus, the set of extensions described here build on top of HPF and concentrate on management of asynchronous tasks and their interaction through SDAs.

2.1 Task Management

Opus tasks are units of coarse-grain parallelism executing in their own address space. Tasks are spawned by explicit activation of *task programs* (entities similar to HPF subroutines, where the keyword TASK CODE replaces SUBROUTINE) using the spawn statement:

SPAWN *task-code-name* (*arg*, ...) **ON** *resource-request*

The semantics of the spawn statement are similar to that of a Unix *fork* in that the spawning task continues execution past the spawn statement, independent (and potentially in parallel) of the newly spawned task. A task *terminates* when its execution reaches the end of the associated task program code, or is explicitly killed.

Tasks execute on a set of system resources specified by the *resource request* in the ON clause of the spawn statement. The resource request consists of a processor specification, used to select a set of processors for nested parallel execution of the task, and a machine specification, used to locate the additional processor(s). Nested parallel execution of the task can occur as task parallelism, by spawning other tasks, or as data parallelism, using HPF specifications. In the latter case, the compiler will generate the appropriate SPMD code to be executed on the selected processors.

2.2 Shared Data Abstractions

*Shared Data Abstraction*s (SDAs) allow independently executing tasks to interact with each other in a controlled yet flexible manner. An SDA specification, modeled after the Fortran 90 *module*, consists of a set of data structures and an associated set of methods (procedures) that manipulate this data. The data and methods can be public or private, where public methods and data are directly accessible to tasks which have access to an instance of the SDA type. Private SDA data and methods can only be used by other data or methods within the SDA. In this respect, SDAs and their methods are similar to C++ classes and class functions [9].

As stated before, access to SDA data is exclusive, thus ensuring that there are no conflicts due to the asynchronous method calls. That is, only one method call associated with an SDA object can be active at any time. Other requests are delayed and the calling task is blocked until the currently executing method completes. In this respect, SDAs are similar to monitors.

Figure 1 presents a code fragment that specifies the SDA *stack* which can be used to communicate integers between tasks in a last-in-first-out manner. The SDA is comprised of two parts, separated by the keyword **CONTAINS**. The first part consists of the internal data structures of the SDA, which in this case have all been declared private, and thus cannot be directly accessed from outside the SDA. The second part consists of the procedure declarations which constitute the methods associated with the SDA.

Each procedure declaration can have an optional *condition clause* using the keyword **WHEN**, which "guards" the execution of the method, similar to Dijkstra's guarded commands [8]. The condition clause consists of a logical expression, comprised of the internal data structures and the arguments to the procedure. A method call is executed only if the associated condition clause is true at the moment of evaluation, otherwise it is enqueued and executed when the condition clause

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

    ...
CONTAINS
  SUBROUTINE get (x) WHEN (count .gt. 0)
    INTEGER x
    x = lifo(count)
    count = count - 1
  end

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

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

Figure 1: Code fragment specifying the *stack* SDA

becomes true. For example, the *put* method can be executed only when *count* is less than *max* and the *get* method can be executed only when *count* is greater than zero.

Similar to HPF procedure declarations, each SDA type may have an optional resource request directive, which allows the internal data structures of the SDA to be distributed across these processors. This is useful (or perhaps necessary) for SDAs that comprise large data structures. The dummy arguments of the SDA methods can also be distributed using the rules applicable to HPF procedure arguments.

In this section, we have briefly described the main features of Opus; a more detailed description of the language features, including SDA type parameterization and persistence, can be found in [4]. A more concrete example using the features of Opus to encode a simplified application for the multidisciplinary design of an aircraft can also be found in [4].

3 Opus Runtime Support

As specified in the previous section, an Opus program is potentially constructed of three types of parallel entities: parallel tasks, data parallel statements, and SDA method tasks. There are several reasons for wanting to map several of these entities onto a single physical processor: to allow for more parallel entities than physical processing elements; to increase locality of SDA data by

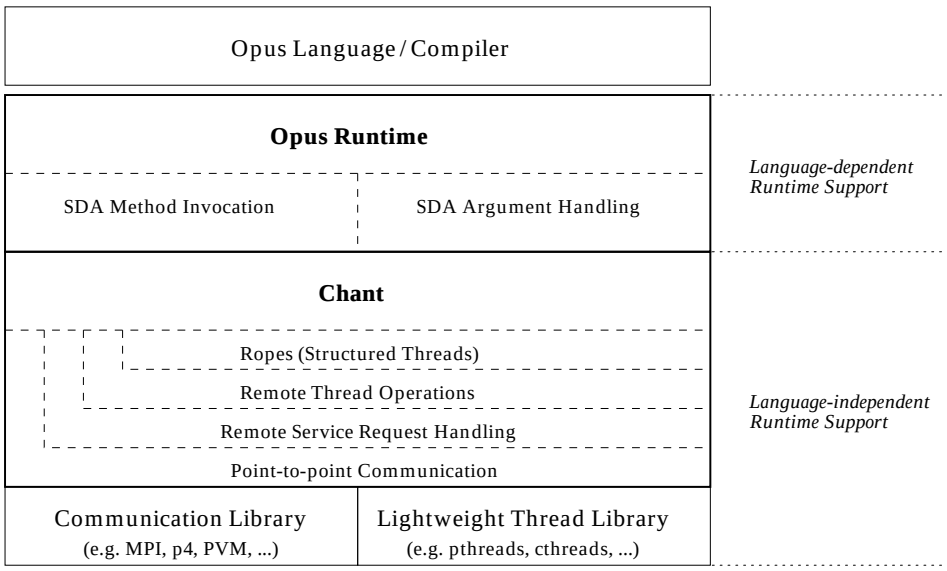

Figure 2: Runtime layers supporting Opus

placing an SDA on the same processor as a parallel task or data parallel statement; or to minimize the idle time for a processor by overlapping the execution of various parallel entities with blocking instructions. To allow for parallel entities that are independent of the physical processing resources, Opus employs the well known concept of a *lightweight thread* to represent its unit of parallelism in all cases. Lightweight threads have several advantages over the traditional approach of using operating system processes for representing parallel tasks: they contain only a minimal context to allow for efficient support of fine-grain parallelism and latency-tolerance techniques; they provide explicit control over scheduling decisions; and they provide the ability to execute tasks on parallel systems that do not support a full operating system on each node [6, 27, 37]. One should note that most of the current lightweight threads packages typically assume a single address space as their execution environment. However, Opus requires underlying support for lightweight threads that are capable of execution and communication in a distributed memory environment.

As depicted in Figure 2, the runtime system is divided into two essential layers: *Opus Runtime*, which provides support for task and SDA management, and *Chant*, which provides underlying support for lightweight threads in a distributed memory environment. Since Chant does not assume any particular knowledge of the Opus language, we refer to this layer as the language-independent layer, whereas Opus Runtime clearly represents the language-dependent layer. This layered approach allows us to build on a large collection of existing runtime work in the areas of lightweight threads and communication systems. In cooperation with other researchers in this field, we are in the process of attempting to define a standard language-independent runtime interface (at the level of Chant) for parallel languages, called *PORTS*. By using a modular design for the Opus runtime system, we are in the position to take advantage of such a system when it emerges.

We now provide a description of both Chant and Opus Runtime, focusing on some of the basic issues that must be addressed to obtain an efficient implementation of the language. Previous work on Chant [17] and Opus Runtime [18] provides a more detailed description of these systems.

3.1 Chant

Despite their popularity and utility in uniprocessor and shared memory multiprocessor systems, lightweight thread packages for distributed memory systems have received little attention. This is unfortunate: in a distributed memory system, lightweight threads can overlap communication with computation (latency tolerance) [10, 16]; they can emulate virtual processors [28]; and they can permit dynamic scheduling and load balancing [5]. However, there is no widely accepted implementation of a lightweight threads package that supports direct communication between two threads, regardless of whether they exist in the same address space or not. Chant has been designed to fill this need.

Chant extends the POSIX pthreads standard for lightweight threads [21] by adding a new object called a `chanter` thread, which supports the functionality for both point-to-point and remote service request communication paradigms, as well as remote thread operations. Point-to-point communication primitives (i.e. *send/receive*) are required to support most existing explicit message passing programs, including those generated by parallelizing compilers [20, 22, 38]. Remote service request communication primitives are needed to support RPC-style communications [29, 34], as well as client-server applications and *get/put* primitives. Remote thread operations are necessary to support threads in a global environment, such as creating a thread on another processor. Ropes, also based on `chanter` threads, are used for implementing parallel computations which are inherently structured, such as SPMD data parallel codes. Ropes provide a an indexed naming scheme for the threads, as well as a scope for global communication, reduction, and synchronization operations.

Chant is built as a layered system (as shown in Figure 2), where point-to-point communication provides the basis for implementing remote service requests, which in turn provides the basis for implementing global thread operations. Our design goals center on portability, based on existing standards for lightweight threads and communication systems, and efficiency, based on supporting point-to-point message passing without interrupts or extra message buffer copies.

3.1.1 Point-to-Point Communication

Point-to-point communication is characterized by the fact that both the sending thread and receiving thread agree that a message is to be transferred from the former to the latter. Although there are various forms of send and receive primitives, their defining aspect is that both sides are knowledgeable that the communication is to occur. As a result of this understanding, it is possible to avoid costly interrupts and buffer copies by registering the receive with the operating system before the message actually arrives. This allows the operating system to place the incoming message in the proper memory location upon arrival, rather than making a local copy of the message in a system buffer. Chant ensures that no message copies are incurred for point-to-point communication that wouldn't otherwise be made by the underlying communication system, which is tantamount to reasonable efficiency on low-latency/high-bandwidth machines.

To support message passing from one thread to another, Chant must provide solutions to the problems of naming global threads within the context of an operating system entity (process), delivering messages to threads within a process, and polling for outstanding messages.

To enable the use of the underlying thread package for all thread operations local to a given

processing element, a `chanter` thread is defined in relation to a particular processing element, which in turn is composed of a process group identifier and a process rank within the group. The group/rank paradigm can be matched to most parallel systems, and also corresponds directly to the MPI standard for representing processing elements [11]. Therefore, a global Chant thread is defined by a 3-tuple, composed of a group identifier, a rank within the group, and a local thread identifier. The type of the local thread identifier is determined by the thread type of the underlying thread package and, although this will vary for different thread packages, allows the global threads to behave normally with respect to the underlying thread package for operations not concerned with global threads.

Most communication systems support delivery to a particular context within a particular processor, but do not provide direct support for naming entities within the context, namely a thread. All message passing systems, however, support the notion of a message header, which is used by the operating system as a *signature* for delivering messages to the proper location (process). Messages also contain a body, which contains the actual contents of the message. In order to ensure proper delivery of messages to threads, and without having to make intermediate copies, the entire global thread name (group, rank, thread) must appear in the message header. Some communication systems, such as MPI, provide a mechanism by which thread names can easily be integrated into the message header. MPI accomplishes this using the *communicator* (i.e. group) field. However, for systems which do not provide support for message delivery to entities within a process, we must overload one of the existing fields in the message header, typically the user-defined tag field. This approach has the disadvantage of reducing the number of tags allowed, but the alternative approach, placing the thread identifier in the message body, would force an intermediate thread to receive all incoming messages, decode the body, and forward the remaining message to the proper thread. In addition to being time consuming, this method would require the message body to be copied for both sending (to insert the thread id) and receiving (to extract the thread id).

The problem of waiting for a message is solved by polling for incoming messages, either by the operating system or user-level code. Operating system polls either block the entire processor while the operating system spins and waits, or allows the process to continue execution and generates an interrupt when a message arrives. Neither of these solutions are desirable, since blocking the entire processor disables the opportunity to execute other (ready) threads, and interrupts are very expensive, causing cache disruptions and mutual exclusion problems. Therefore, in Chant, message polling does not involve the operating system and is done either by the thread which initiates the receive request or by the scheduler. The *thread-polls* method has the advantage of not having to register receive operations with the scheduler, and will work for any underlying thread package, but will cause context switching overheads in the case when a thread is re-scheduled but cannot complete the receive operation. The *scheduler-polls* method requires all threads to register a receive with the scheduler, and then are removed from the ready queue and placed on a blocking queue until the message arrives. This avoids the overhead of scheduling a thread that is not really ready to run, but forces the scheduler to poll for outstanding messages on each context switch, and may not be supported by some lightweight thread libraries. An analysis of these policies is given in [17].

An initial implementation of the Chant point-to-point communication layer for the Intel Paragon was found to introduce only a 6% overhead (on average) for a 1K byte message when compared to the same operations using Paragon processes [17]. These initial performance numbers are encouraging as they demonstrate that thread-based communication introduce little overhead.

3.1.2 Remote Service Requests

Having established a mechanism by which lightweight threads located in different addressing spaces can communicate using point-to-point mechanisms, we now address the problem of supporting remote service requests, which builds on our design for point-to-point message passing. Remote service request messages are distinguished from point-to-point messages in that the destination thread is not expecting the message. Rather, the message details some request that the destination thread is to perform on behalf of the source thread. The nature of the request can be anything, but common examples include returning a value from a local addressing space that is wanted by a thread in a different addressing space (*remote fetch*), executing a local function (*remote procedure call*), and processing system requests necessary to keep global state up-to-date (*coherence management*).

Most remote service requests require some acknowledgment to be sent back to the requesting thread, such as value of a remote fetch or the return value from a remote procedure call. To minimize the amount of time the source thread remains blocked, we wish to process the remote service request as soon as possible on the destination processor, but without having to interrupt a computation thread prematurely. Interruptions are costly to execute and can disrupt the data and code caches which, as processor states increase, will continue to have a detrimental effect on the efficiency of a program [29]. Also, the MPI standard [11] does not support interrupt-driven message passing, thus utilizing interrupts in a design would preclude the use of the MPI communications layer. Therefore, we need a polling mechanism by which remote service requests can be checked without having to prematurely interrupt a computation thread when such a request arrives.

Since the main problem with remote service requests is that they arrive at a processor unannounced, we simply introduce a new thread, called the *server thread*, which is responsible for receiving all remote service requests. Using one of the polling techniques outlined in Section 3.1.1, the server thread repeatedly issues nonblocking receive requests for any remote service request message, which can be distinguished from point-to-point messages by virtue of being sent to the designated server thread rather than a computation thread. When a remote service request is received, the server thread assumes a higher scheduling priority than the computation threads, ensuring that it is scheduled at the next context switch point.

If the underlying architecture supports a low-latency remote service request mechanism, such as Active Messages [34], in addition to the point-to-point primitives, then Chant would ideally shortcut the remote service request mechanism just described to take advantage of these primitives.

3.1.3 Remote Thread Operations

As well as adding communication primitives to a lightweight thread interface, Chant must support the existing lightweight thread primitives that are inherited from the underlying thread package. These primitives provide functionality for thread management, thread synchronization, thread scheduling, thread-local data, and thread signal handling. We divide these primitives into two groups: those affected by the addition of global thread identifiers in the system, and those not affected. For example, the thread creation primitive must be capable of creating remote threads, but the thread-local data primitives are only concerned with a particular local thread. Our goal in designing Chant is to provide an integrated and seamless solution for both groups of primitives.

This is accomplished in two ways.

1. Since a global thread identifier contains a local thread id, it is possible to extract this identifier (using the `pthread_chanter_thread` primitive) and use it as an argument for thread primitives concerned only with local threads, such as manipulating the thread-specific data of a local thread.

2. Thread primitives that are affected by the global identifiers either take a thread identifier as an argument (such as join) or return a thread identifier (such as create). In either case, the primitive must handle the situation of a thread identifier that refers to a remote thread.

Having described the details of how Chant supports remote service requests, we can now utilize this functionality in the form of a remote procedure call. Similar to the manner in which Unix creates a process on a remote machine [35], Chant utilizes the server thread and the remote service request mechanism to implement primitives which may require the cooperation of a remote processing element.

3.1.4 Ropes

Threads represent task parallelism and the general concept of MIMD programming, where one thread is created for each parallel task that is desired, and execution of the threads is independent and asynchronous. However, data parallelism represents a particular specialization of MIMD programming (often termed SPMD programming), in which all tasks execute the same code but on separate data partitions. Therefore, what is required is a group of threads that captures this behavior better than a collection of otherwise unrelated threads. Specifically, a *rope* is defined as a collection of threads, each executing the same code segment, and participating in certain global communication and coordination operations, where the scope of these global operations is defined as the threads within the rope. This allows for data parallel tasks to mix with unrelated tasks on the same processor, yet restrict all global communication and synchronization operations to those threads that are participating in the data parallel task.

Chant supports ropes by introducing a new object, called a `rope`, which corresponds to a data parallel thread. In addition to the attributes of a global thread, a `rope` thread contains a index relative to the threads within that rope so that all threads within a rope may refer to each other using this index rather than the global thread identifier. This rope index naming scheme is maintained using a local translation table on each processor, which translates a rope identifier and index into a global thread identifier. In addition, new primitives for collective communication and reduction are available to `rope` threads.

3.2 Opus Runtime

In the last subsection, we presented Chant, the language-independent part of the runtime system. We now describe the design of the runtime system required to support the features of Opus that differ from HPF: tasks and SDAs. The two major issues in the Opus runtime system are the management of the execution and interaction of tasks and SDAs. In the initial design, we have

concentrated on the interaction and have taken a simplified approach to resource management. We presume that all the required resources are statically allocated and the appropriate code is invoked where necessary. We will later extend the runtime system to support the dynamic acquisition of new resources.

The interaction between tasks and SDAs requires runtime support for both method invocation and method argument handling. These issues are explored in the following subsections.

3.2.1 SDA Method Invocation

The semantics of SDAs place two restrictions on method invocation:

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

We can view an SDA as being comprised of two components: a control structure, which executes the SDA methods in accordance with the stated restrictions, and a set of SDA data structures.

At this point, our design only supports a centralized SDA control structure, represented by a single *master* thread on a specified processor. All remaining SDA processors will host *worker* threads, which take part in the method execution when instructed by the master thread. The first restriction, mutually-exclusive access to the SDA, is guaranteed by the fact that the SDA control structure is centralized. Allowing for distributed control of an SDA would require implementing distributed mutual exclusion algorithms, such as [24], to guarantee the monitor-like semantics of SDAs, and is a point of interest for future research.

Having established the master-worker organization of the SDA control structure, we can now describe a simple mechanism for ensuring that the second restriction, conditional execution of the methods, is enforced. When an SDA master thread receives a request to execute a method, its condition function is first evaluated to see if the condition is true and, if not, the method is enqueued and another request is handled. Whenever a condition function evaluates to true, the associated method is invoked, after which the condition functions for any enqueued methods are examined to see if their conditions have changed. Starvation is prevented by ensuring that any enqueued method whose condition has changed, is processed before a new method request is handled.

Details of a prototype implementation of this method invocation mechanism, along with preliminary experiments measuring the overhead of the method invocation design can be found in [18]. Our initial experiments show that our design adds only a small amount of overhead to a raw RPC call across a network of workstations.

3.2.2 SDA Method Argument Handling

Opus supports the exploitation of data parallelism within tasks. For example, an Opus task may be an HPF code with the data structures distributed across a set of processors. On the other

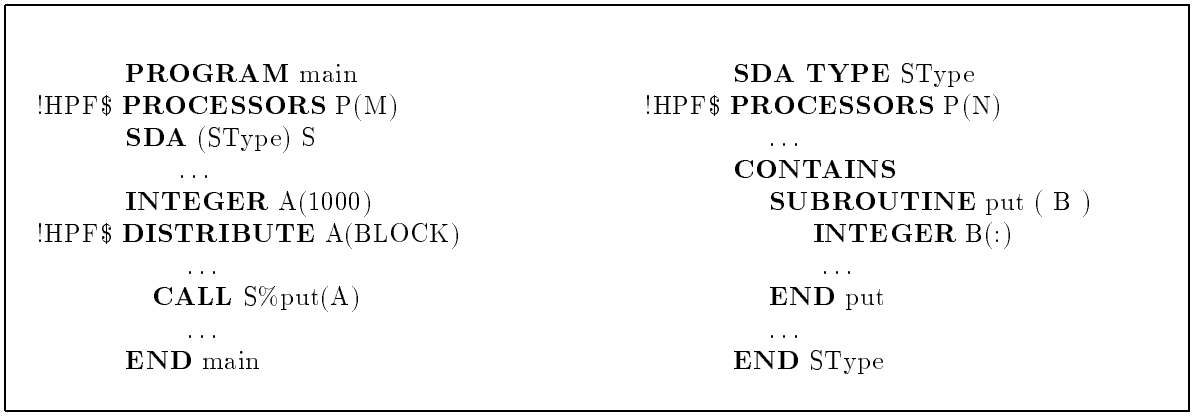

Figure 3: SDA code excerpt

hand, SDAs may also embody data parallelism and have their internal data structures mapped to a set of processors. In the most general scenario, a task, and the SDA with which it is interacting, will be executing on independent set of resources. Thus, the method arguments will have to be communicated and remapped before the method code is executed. In this section, we discuss the issues of handling distributed method arguments.

To illustrate the issues, let's consider the code excerpt in Figure 3, which declares an HPF computation, `main`, on `M` processors, and an SDA, `S`, distributed on `N` processors. The task `main` contains an array, `A`, that is distributed by BLOCK across the `M` processors. At some point, the values from the distributed array `A` are used to update the SDA array, `B`, using the SDA method `put`. Let us consider the issues that arise with different values of `M` and `N`.

If `M` and `N` are both greater than 1, then both `main` and `S` along with their data structures are distributed. We will assume that `main` and `S` are each represented by a set of threads distributed over the processors, and that each contains a *master* thread among the set, which is responsible for external coordination of the thread group. To execute the `put` method, we have the following options for transferring the data from `A` (the actual argument) to `B` (the formal argument):

1. The master thread from `main` collects the elements of `A` into a local scratch array, then sends it to the master thread for `S`, which distributes the values among `S`'s remaining threads, such that each thread updates its portion of `B`. This provides the simplest solution in terms of scheduling data transfers, since only one transfer occurs, from master thread of `main` to master thread of `S`. However, two scratch arrays and two gather/scatter operations are required, consuming both time and space.

2. The master thread from `main` collects the elements of `A` into a local scratch array, then negotiates with the master thread of `S` to determine how the scratch array is to be distributed among the threads of `S`, taking `B`'s distribution into account. After the negotiation, `main`'s master thread distributes the scratch array directly to `S`'s threads. This approach eliminates one scratch array and the scatter operation, but introduces a negotiation phase that is required to discern `B`'s distribution.

3. The master thread from `main` negotiates with the master thread of `S`, then informs the other

threads in `main` to send their portion of `A` to `S`'s master thread. When the master thread from `S` has received all of the messages and formed a local scratch array, the array is distributed among the remaining threads in `S`. As with the previous scenario, this approach eliminates a one scratch array and a gather operation at the expense of a negotiation phase.

4. The master thread from `main` negotiates with the master thread of `S`, then informs the other threads in `main` to send their portion of `A` to the appropriate threads in `S`, according to the distribution of `B`. This approach eliminates both scratch arrays and gather/scatter operations, but requires all threads from `S` and `main` to understand each others array distribution.

This level of complexity in data structure management is necessary to accommodate the various modules which comprise an Opus code, since each module will typically be developed independently of the others and may view the same data in a different format or distribution. In addition to remapping a data structure from one distribution to another, the SDA may be required to change the dimensionality of a data structure or to filter the data using some predefined filter. The methods outlined above will accommodate all of these requests.

The choice of which method to use will be dependent on various factors and may sometimes be made statically at compile time. However, we presume that in most cases the negotiation will occur at runtime and the choice made based on the current set of circumstances. We have started an initial implementation of the above design and will experiment with the various choices in order to characterize the overheads involved with each of them.

4 Related Research

Other language projects that focus on the integration of task and data parallelism include Fortran-M [12, 13], DPC [30] and FX [31, 32]. Fortran-M and DPC support mechanisms for establishing message pathways between tasks, and all communication between the tasks is transmitted via these pathways. Fortran-M uses the concept of "channels" (similar to a Unix pipe [1]) for establishing its pathways, and DPC uses concepts similar to C file structures. In FX, tasks communicate only through arguments at the time of creation and termination. In contrast, Opus allows communication between tasks in the form of SDAs, which are data structures that resemble C++ objects with monitor-like semantics and conditions placed on method invocation.

Thread-based runtime support for parallel languages is common in the realm of shared memory multiprocessors [2, 15, 23, 25, 26, 36]. One lightweight thread system for shared memory multiprocessors, *pthreads++*, supports the notion of ropes for data parallel execution of threads [33]. However, lightweight thread systems for distributed memory multiprocessors are far less common. In addition to several application-specific runtime systems that support distributed memory threads in a rather ad-hoc fashion [7, 16, 28], there are two systems which provide a general interface for lightweight threads capable of communication in a distributed memory environment: Nexus and NewThreads. Nexus [14] supports a remote service request mechanism between threads, based the asynchronous remote procedure call provided in Active Messages [34]. However, standard send/receive primitives are not directly supported, and the overhead required to provide this functionality atop a remote service request mechanism is currently unknown. In addition, the cost of

selecting, verifying, and calling the correct message handling routine without hardware and operating system support is expensive on most machines [29, 34]. Chant takes the opposite approach by providing a basis for efficient point-to-point communication (using well-known libraries), on top of which a remote service request mechanism is provided. NewThreads provides communication between remote threads using a "ports" naming mechanism, where a port can be mapped onto any thread, and thus requires a global name server to manage the unique port identifiers. Chant uses a thread identifier that is relative to a given process and processor, thus eliminating the need for a global name server for basic point-to-point communication, and also supports the POSIX standard pthreads interface for the lightweight thread operations.

5 Conclusions

Opus extends the HPF standard to facilitate the integration of task and data parallelism. In this paper we have provided an overview of the runtime system necessary to support the new extensions on a variety of parallel and distributed systems. The runtime system is divided into a language-dependent layer atop a language-independent layer. The language-dependent layer, called Opus Runtime, provides the functionality necessary for supporting tasks and SDAs. The language-independent layer, called Chant, provides lightweight threads that support point-to-point communication, remote service requests, and remote thread operations in a distributed memory environment. In addition, Chant provides a mechanism for grouping threads into collections called ropes for the purpose of performing data parallel operations. Chant is built atop standard lightweight thread and communication libraries, and adds relatively little overhead to either of these layers. In addition to supporting Opus Runtime, Chant can provide support for other projects which can benefit from the use of lightweight threads, such as distributed simulations and load balancing.

Preliminary experiments with various parts of the Opus runtime system indicate that the overhead introduced by our design is fairly small. We are currently enhancing our prototype and will report on the performance of Opus Runtime and Chant in future papers.

References

[1] Maurice J. Bach. *The Design of the UNIX Operating System.* Software Series. Prentice-Hall, 1986.

[2] Brian N. Bershad, Edward D. Lazowska, Henry M. Levy, and David B. Wagner. An open environment for building parallel programming systems. Technical Report 88-01-03, Department of Computer Science, University of Washington, January 1988.

[3] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[4] Barbara M. Chapman, Piyush Mehrotra, John Van Rosendale, and Hans P. Zima. A software architecture of multidisciplinary applications: Integrating task and data parallelism. ICASE Report 94-18, Institute for Computer Applications in Science and Engineering, Hampton, VA, March 1994. To appear in CONPAR 94.

[5] T. C. K. Chou and J. A. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, SE-8(4), July 1982.

[6] Cray Research, Inc. *Cray T3D System Architecture Overview*, 1994.

[7] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In 4^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[8] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[9] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990. ISBN 0-201-51459-1.

[10] Edward W. Felton and Dylan McNamee. Improving the performance of message-passing applications by multithreading. In *Scalable High Performance Computing Conference*, pages 84–89, April 1992.

[11] Message Passing Interface Forum. *Document for a Standard Message Passing Interface*, draft edition, November 1993.

[12] I. Foster, M. Xu, B. Avalani, and A. Choudhary. A compilation system that integrates High Performance Fortran and Fortran M. In *Scalable High Performance Computing Conference*, May 1994.

[13] I. T. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. Technical Report MCS-P327-0992 Revision 1, Mathematics and Computer Science Division, Argonne National Laboratory, June 1993.

[14] Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke. Nexus: An interoperability layer for parallel and distributed computer systems. Technical Report Version 1.3, Argonne National Labs, December 1993.

[15] Dirk Grunwald. A users guide to AWESIME: An object oriented parallel programming and simulation system. Technical Report CU-CS-552-91, Department of Computer Science, University of Colorado at Boulder, November 1991.

[16] Matthew Haines and Wim Böhm. An evaluation of software multithreading in a conventional distributed memory multiprocessor. In *IEEE Symposium on Parallel and Distributed Processing*, pages 106–113, December 1993.

[17] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. ICASE Report 94-25, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681, April 1994.

[18] Matthew Haines, Bryan Hess, Piyush Mehrotra, John Van Rosendale, and Hans Zima. Runtime support for data parallel tasks. ICASE Report 94-26, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681, April 1994.

[19] High Performance Fortran Forum. *High Performance Fortran Language Specification*, version 1.0 edition, May 1993.

[20] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

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

[22] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[23] Jeff Kramer, Jeff Magee, Morris Sloman, Naranker Dulay, S.C. Cheung, Stephen Crane, and Kevin Twindle. An introduction to distributed programming in REX. In *Proceedings of ESPRIT-91*, pages 207–222, Brussels, November 1991.

[24] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[25] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX*, pages 29–41, San Diego, CA, January 1993.

[26] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A machine independent interface for lightweight threads. Technical Report CIT-CC-93/53, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1993.

[27] nCUBE, Beaverton, OR. *nCUBE/2 Technical Overview, SYSTEMS*, 1990.

[28] David M. Nicol and Philip Heidelberger. Optimistic parallel simulation of continuous time markov chains using uniformization. *Journal of Parallel and Distributed Computing*, 18(4):395–410, August 1993.

[29] Matthew Rosing and Joel Saltz. Low latency messages on distributed memory multiprocessors. Technical Report ICASE Report No. 93-30, Institute for Computer Applications in Science and Engineering, NASA LaRC, Hampton, Virginia, June 1993.

[30] B. Seevers, M. J. Quinn, and P. J. Hatcher. A parallel programming environment supporting multiple data-parallel modules. In *Workshop on Languages, Compilers and Run-Time Environments for Distributed Mmeory Machines*, October 1992.

[31] J. Subhlok and T. Gross. Task parallel programming in Fx. Technical Report CMU-CS-94-112, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.

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

[33] Neelakantan Sundaresan and Linda Lee. An object-oriented thread model for parallel numerical applicaitons. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, pages 291–308, Sunriver, OR, April 1994.

[34] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communications and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[35] W. E. Weihl. Remote procedure call. In Sape Mullender, editor, *Distributed systems*, chapter 4, pages 65–86. ACM Press, 1989.

[36] Mark Weiser, Alan Demers, and Carl Hauser. The portable common runtime approach to interoperability. *ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989.

[37] Stephen R. Wheat, Arthur B. Maccabe, Rolf E. Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, Maui, HI, January 1994.

[38] Hans P. Zima and Barbara M. Chapman. Compiling for distributed memory systems. *Proceedings of the IEEE, Special Section on Languages and Compilers for Parallel Machines (To appear 1993)*, 1993. Also: Technical Report ACPC/TR 92-16, Austrian Center for Parallel Computation.