Runtime Support for Data Parallel Tasks[*]

Matthew Haines[†] Bryan Hess[†] Piyush Mehrotra[†] John Van Rosendale[†]
Hans Zima[‡]

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

[‡]Institute for Software Technology and Parallel Systems
University of Vienna
Brünner Strasse 72, A-1210, Vienna, Austria
zima@par.univie.ac.at

April 27, 1994

Abstract

We have recently introduced a set of Fortran language extensions that allow for integrated support of task and data parallelism, and provide for shared data abstractions (SDAs) as a method for communication and synchronization among these tasks. In this paper we discuss the design and implementation issues of the runtime system necessary to support these extensions, and discuss the underlying requirements for such a system. To test the feasibility of this approach, we implement a prototype of the runtime system and use this to support an abstract multidisciplinary optimization (MDO) problem for aircraft design. We give initial results and discuss future plans.

1 Introduction

Most of the recent research effort in parallel languages and compilers has concentrated on specification and exploitation of data parallelism in scientific codes. However, there are a large number of scientific and engineering codes which exhibit multiple levels of parallelism. Multidisciplinary applications are a good example of such codes. These applications, such as weather modeling and aircraft design, integrate codes from different disciplines to solve a larger and more complex problem. In general, 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. For example, the design of an aircraft requires data parallel codes from disciplines such as aerodynamics, propulsion, structural analysis, controls and so forth to interact asynchronously while optimizing the design variables of the aircraft.

[*]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.

Data parallel language extensions such as High Performance Fortran (HPF) [17] and Vienna Fortran [5] are adequate for the parallelism within individual discipline codes, but do not provide any support for coordinating the execution and interaction of these codes. We have recently designed a set of extensions to HPF which provide such support [6]. Along with extensions to manage independently executing tasks, we have introduced a new mechanism, called *Shared Data Abstractions* (SDAs), to allow these tasks to share data with each other. SDAs generalize Fortran 90 modules by including features from both *objects* in object-oriented systems and *monitors* in shared memory languages. This provides a high-level, controlled and clean interface for large grained parallel tasks to interact with each other in a plug compatible manner.

In this paper, we concentrate on the design of a runtime support system to address the specific needs of SDAs. In particular, we address two specific areas of design for SDA runtime support: distributed data structure management and SDA method invocation (see Figure 5). Distributed data structure management includes the distribution of SDA data structures as well as the protocols for interfacing with distributed data structures in the individual task codes. Method invocation includes the protocols for accessing and executing SDA method functions based on the monitor semantics and condition clauses that can guard every SDA method. The SDA runtime system is based on lightweight, user-level threads that are capable of supporting both intra-processor and inter-processor communication primitives in the form of shared memory, message passing, and remote service requests [16]. This allows the independently executing tasks and the SDA methods to share the underlying parallel resources. Along with asynchronous communication between threads, the system also supports collective communication among SPMD threads executing in a loosely synchronous manner, such as those that might be produced by an HPF compiler for data parallel computation. Since data parallel tasks need to communicate distributed data to each other, the runtime system also supports communication between two sets of SPMD threads.

Other projects that focus on the integration of task and data parallelism include Fortran-M [11, 12], DPC [27] and FX [28, 29]. Fortran-M and DPC support mechanisms for establishing message plumbing between tasks, directly based on ports and an extension of C file structures, respectively. In Fx, tasks communicate only through arguments at the time of creation and termination. The runtime support systems for these efforts face some of the same issues discussed in this paper. However, unlike the SDA runtime system described here, these systems are not based on lightweight threads.

The remainder of the paper is organized as follows: Section 2 summarizes the Fortran language extensions for supporting both task parallelism and shared data abstractions; Section 3 outlines the runtime support necessary for supporting these extensions, with particular respect to data distribution and method invocation issues; and Section 4 introduces a prototype of the SDA runtime system, developed to test the feasibility of the SDA method invocation design.

2 An Introduction to Shared Data Abstractions

In this section, we provide a short overview of the Fortran extensions we have designed to support the integration of task and data parallelism. Full details of the these extensions can be found in [6].

In our system, 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 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, which 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.

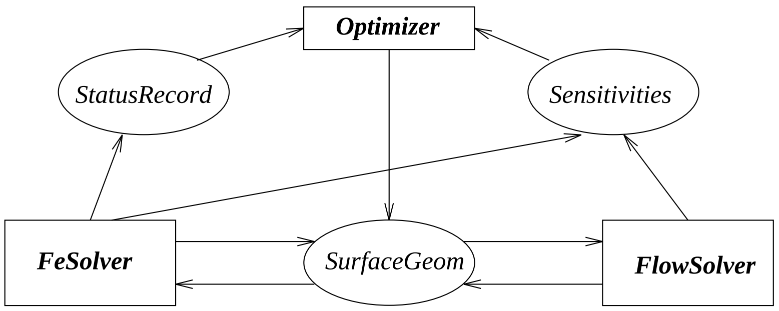


Figure 1: A sample MDO application for aircraft design

We presume that High Performance Fortran (HPF) [17] 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. Multidisciplinary optimization (MDO) problems form a natural target for integrating task and data parallelism, as they are commonly formed by combining data parallel units from various disciplines to form a single application. As an example, we introduce a simplified MDO application for aircraft design that will highlight the need for supporting task parallel constructs and communication between tasks using data structure repositories (SDAs).

2.1 A Sample MDO Application

We now briefly describe a focus application: the simultaneous optimization of the aerodynamic and structural design of an aircraft configuration. Though a realistic multidisciplinary optimization of a full aircraft configuration would require a number of other discipline codes, such as controls, we present this simplified version for the sake of brevity.

The structure of the program is shown in Figure 1 where the tasks are represented by rectangles and the SDAs by ovals. The *Optimizer* is the main task and controls the execution of the entire MDO application. It creates three SDAs (*StatusRecord*, *Sensitivities*, and *SurfaceGeom*) and makes them available to the other two tasks, *FlowSolver* and *FeSolver*, as they are spawned.

The *Optimizer* initiates execution of the application by storing the initial geometry in the SDA *SurfaceGeom*. The *FeSolver* task generates a finite element model based on this geometry and uses some initial forces to determine the structural deflections. These deflections are then stored in *SurfaceGeom* as a deflected geometry. The *FlowSolver*, meanwhile, generates an aerodynamics grid based on the initial geometry and performs an analysis of the airflow around the aircraft, producing a new flow solution.

In subsequent cycles, the *FeSolver* uses forces based on the current flow solution to produce new deformations, while the *FlowSolver* uses the deformed geometry and the previous flow solution to produce new solutions. This process is repeated until the *FeSolver* determines that the difference between the new and old deflections is within some specified tolerance. At this point, it places some output variables in the *StatusRecord* SDA. Both *FeSolver* and *FlowSolver* also produce sensitivity derivatives and place them in the *Sensitivities* SDA. These represent the behavior of the output variables, such as lift and drag, with respect to design variables, such as the sweep angle of the wing.

When, the inner iteration is completed, the *Optimizer* obtains the output variables and the sensitivity derivatives and, based on some objective function that it is minimizing, decides whether to terminate the program or to produce a modified base geometry. In the latter case, it places the new geometry in the *SurfaceGeom* SDA, and the inner cycle is repeated.

```
SPAWN FlowSolver (SurfaceGeom, Sensitivities,...) ON  resource-request
```

Figure 2: Code fragment spawning the *FlowSolver* task

2.2 Task Management

We now describe the HPF extensions required to create and manage tasks. These tasks are units of coarse-grain parallelism executing in their own address space. They are spawned by explicit activation of *task programs*, entities syntactically similar to a Fortran subroutine (except for the keyword **TASK CODE** which is used instead of **SUBROUTINE**). The spawn is non-blocking in that the spawning task continues its execution after the spawn. A task *terminates* when its execution reaches the end of the associated task program code, or if it is explicitly killed.

An example of how the *Optimizer* task could spawn the *FlowSolver* task is depicted in Figure 2, where *SurfaceGeom* and *Sensitivities* are SDA objects which are passed as arguments to the *FlowSolver* task.

Tasks operate on a set of system resources allocated to them at the time of their spawning, either through an explicit resource request (using an *on clause* as shown above) or as defaults assigned by the system. A resource request has two optional parts: a machine specification and a processor specification. The machine specification can be used to specify a particular physical machine or a class of machines. If a class of machines is specified, such as *Sun Sparc 10*, then the system is free to choose one from a set of such machines. The processor specification is used to select the set of processors on which the specified task will execute. In either case, the user can request that the spawned task use part of the spawning task's resources, or that the system should allocate new resources for the spawned task.

Tasks may have nested functional or data parallelism, where the former is embodied by spawning other tasks, and the latter is specified using HPF directives. Thus, the task code specification may include a **PROCESSORS** directive along with directives that specify the distribution of data across these processors. It is then the compilers job to produce the appropriate SPMD code for the data parallel task.

2.3 Shared Data Abstractions

An SDA type specification, modeled after the Fortran 90 *module* [2], 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 an SDA. In this respect, SDAs and their methods are similar to C++ classes and class functions [8].

As stated before, access to SDA data is exclusive, thus ensuring that there are no data 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 blocked until the currently executing method completes.

Figure 3 presents a code fragment that depicts a portion of the type specification for the *SurfaceGeom* SDA. The first part (before the keyword **CONTAINS**) consists of the internal data structures of the SDA, all of which have been declared private here, and thus cannot be directly accessed from outside. The second part (after the keyword **CONTAINS**) consists of the procedure declarations which constitute the methods associated with the SDA.

Each procedure declaration can have an optional *condition clause* which "guards" the execution of the method, similar to Dijkstra's guarded commands [7]. The condition clause consists of a logical expression,

```
SDA TYPE SGeomType (StatusRecord)
   SDA (StatRecType) StatusRecord
   TYPE (surface) base
   TYPE (surface) deflected
      ...
   LOGICAL DeflectFull = .FALSE.
   PRIVATE base, deflected, DeflectFull
      ...

CONTAINS
   SUBROUTINE PutBase (b)
      TYPE (surface) b
      base = b
      deflected = b
      DeflectFull = .TRUE.
        ...
   END

   SUBROUTINE GetDeflected (d) WHEN DeflectFull
      TYPE (surface) d
      DeflectFull = .FALSE.
      d = deflected
   END
    ...

 END SurfaceGeom
```

Figure 3: Code fragment specifying for the *SurfaceGeom* SDA

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. If the condition clause evaluates to false, the corresponding method call is enqueued until the expression evaluates to true, as a result of the SDA data being modified by another method call. A method that is declared without a condition clause will be assigned a default condition clause that always evaluates to true. For example, in the above code, the calls to method *GetDeflected* will be executed only when *DeflectFull* is true. Thus, if a task calls *GetDeflected* before a call for *PutBase*, the former is blocked until the latter is executed, ensuring that the variable *deflected* has an appropriate value. Condition clauses, therefore, provide a way to synchronize task interactions based on data dependencies.

Similar to HPF procedure declarations, each SDA type may have an optional *processors* directives 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 an HPF procedure.

An SDA variable is declared using syntax similar to Fortran 90 derived types. For example, in our MDO application the *Optimizer* might declare the SDA variables as shown in Figure 4, where *%* is the operator used for member selection. As with any other global variable, an SDA variable has to be initialized before it can be used. This is done using the predefined methods *INIT* (as shown in Figure 4), which initializes the SDA by allocating the internal data structures from the heap, or *LOAD*, which loads the data from secondary storage. The corresponding *SAVE* method can be used by the programmer to save the internal state of an SDA to secondary storage for later use. This allows SDAs to be *persistent*, which is an important consideration for most MDO applications. An SDA may specify an optional resource request, similar to the one used when spawning tasks, which is used to specify the resources to be used for executing the SDA.

Having provided an initial overview of tasks and SDAs, we now examine the issues involved with providing runtime support for the SDA extensions. A detailed description of the task and SDA extensions,

```
SDA (StatRecType) StatusRecord
SDA (SGeomType) SurfaceGeom
SDA (SensType) Sensitivities

    CALL StatusRecord%INIT
    CALL SurfaceGeom%INIT(StatusRecord)
    CALL Sensitivities%INIT
     ...
```

Figure 4: Declaration and initialization of SDA variables.

including sample code for a similar MDO application, can be found in [6]

3 SDA Runtime Support

If we take an abstract view of the SDA problem, we see that there are two basic types of "tasks" that must be mapped to a set of physical resources: computation tasks, responsible for executing the actual computations being performed, and SDA tasks, responsible for executing the SDA methods and performing any resulting communication operations. Each processor participating in a computation will be assigned at least one computation task, and each processor participating in the storage of an SDA object will be assigned at least one SDA task. Since multiple computations and SDAs may utilize the same (or overlapping) resources, any given processor in the system might be responsible for the simultaneous execution of multiple, independent tasks. Execution of these multiple tasks can be implemented on Unix-based systems by mapping each task to a process, where each processor can execute multiple processes in some fashion. However, this process-based approach has several drawbacks, including

- the inability to control scheduling decisions, since Unix processes are scheduled by the operating system with little input from the user;
- the inability to share addressing spaces between tasks, since each Unix process is assigned its own address space. It is desirable for the SDA task to deliver data to the computation task without involving the computation task;
- costly context switching, due to the large amount of context associated with a (*heavyweight*) Unix process; and
- the inability to execute on systems that do not fully support multiple Unix processes per processor, such as the nCUBE/2 [24], or systems running special microkernels without process support, such as the SUNMOS kernel for the Paragon.

In light of the disadvantages of mapping our tasks to Unix processes, our approach is to utilize lightweight, user-level threads to represent these various independent tasks. A lightweight, user-level thread is a unit of computation with minimal context that executes within the domain of a kernel-level entity, such as a Unix process or Mach thread. Lightweight threads are becoming increasingly useful in supporting language implementations for both parallel and sequential machines by providing a level of concurrency within a kernel-level process. Threads are used in simulation systems [14, 26] to provide parallel events that can be scheduled on a single processor, language implementations [20, 22, 25] to provide support for coroutines, Ada tasks, or C++ method invocations, and generic runtime systems [13, 15, 32] to support fine-grain parallelism and multithreading capabilities. Additionally, the POSIX committee has adopted a

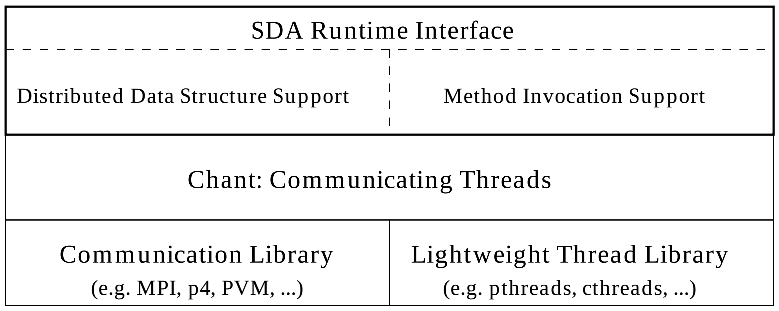

SDA Runtime Interface
Distributed Data Structure Support
Method Invocation Support
Chant: Communicating Threads
Communication Library
(e.g. MPI, p4, PVM, ...)
Lightweight Thread Library
(e.g. pthreads, cthreads, ...)

Figure 5: Runtime layers for SDA support

standard for a lightweight threads interface [18], and many lightweight thread libraries have been designed and implemented for workstations and shared memory multiprocessors [1, 3, 9, 19, 23, 30].

Lightweight threads offer significant advantages over heavyweight processes, including full control over thread scheduling, shared addressing spaces among threads within the same process, very fast context switching, and the ability to execute on systems that do not provide multiprocess support. The main drawbacks of using threads to support our tasks are a lack of portability and standardization, and a lack of support for distributed memory communication primitives. The computation tasks will require point-to-point communication primitives and SDA tasks will require both point-to-point and remote service request primitives to communicate with tasks located in other memory spaces. For example, the computation task may be an HPF module that contains send/receive primitives inserted by the compiler, and an SDA task may need to execute a remote service request to fetch a piece of remote data. Our solution to these problems is to implement our SDA runtime support, as depicted in Figure 5, atop a runtime interface called *Chant* [16] that we are currently developing. Chant supports both a standardized interface for thread operations (as specified by the POSIX thread standard [18]) and communication among threads using either point-to-point primitives (such as those defined in the MPI standard [10]) or remote service requests (such as Active Messages [31]). A description of Chant, and its current status, can be found in [16].

Figure 5 depicts the SDA runtime system as being composed of two portions: one for distributed data structure management and one for method invocation management. In the next two sections we examine these portions in more detail.

3.1 Distributed Data Structures

In addition to the two types of threads being potentially mapped to each processor (computation threads and SDA threads), there are two types of data structures that must be distributed across the processors: computation data, which is any data structure defined within a computation task, and SDA data, which is any data structure defined within an SDA. Since each type of data may be independently distributed over a set of processor memories, we must have a mechanism for transferring data between the two. We now illustrate the issues that arise when data must be transferred from a distributed computation data structure to a distributed SDA data structure. To illustrate the point, let's consider the code excerpt in Figure 6, which declares an HPF computation, `main`, on `M` processors, and an SDA, `S`, distributed among `N` processors. The task `main` contains an array, `A`, that is distributed by BLOCK across the same `M` processors that contain `main`. At some point, the values from the distributed array `A` are used to update the SDA array, `B`, using the SDA method `put`. Let's 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 may be responsible for external coordination of

```
        PROGRAM main                                SDA TYPE SType
!HPF$   PROCESSORS P(M)                     !HPF$   PROCESSORS P(N)
        SDA (SType) S                               ...
          ...                                       CONTAINS
        INTEGER A(1000)                               SUBROUTINE put ( B )
!HPF$   DISTRIBUTE A(BLOCK)                             INTEGER B(:)
          ...                                             ...
          CALL S%put(A)                               END put
          ...                                       ...
        END main                                    END SType
```

Figure 6: SDA code excerpt

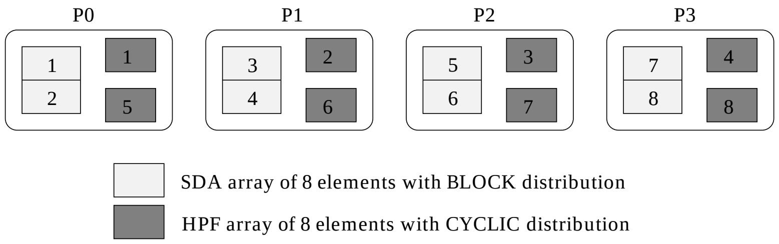

P0
P1
P2
P3
1
2
1
5
3
4
2
6
5
6
3
7
7
8
4
8
SDA array of 8 elements with BLOCK distribution
HPF array of 8 elements with CYCLIC distribution

Figure 7: Sample array distributions for HPF and SDA tasks

the thread group. To execute the `put` method, we have the following options for transferring the data from `A` to `B`:

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`, considering `B`'s distribution. 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.

The complications in passing data from a computation task to an SDA are a result of the different ways a particular data structure may want to be viewed. For example, consider the situation in Figure 7, where an SDA array is distributed differently than an HPF array, and we wish to update the SDA array with the values from the HPF array. To avoid scratch arrays and gather/scatter operations among the HPF and SDA threads, each HPF thread must know which array elements it owns locally, and where the corresponding SDA array elements are located. Consider the HPF computation thread on `P1`, which owns elements 2 and 6 of the HPF array. After negotiations, it learns that element 2 of the SDA array is located on processor `P0` and element 6 of the SDA array is located on processor `P2`, and can then send the array elements to the processors directly.

Referring again to our example in Figure 6, we explore the special case of having either `M` or `N` (or both) be restricted to 1. When `N` is 1, the SDA is executed on a single processor, and all computation threads will send their array values to the single SDA processor. Likewise, when `M` is 1, the computation array is stored on a single processor, and so a single computation thread will distribute the array to the SDA processors. When both `M` and `N` are 1, at most a single message is needed to update the SDA array. We can summarize the support for distributed SDA data structures as follows:

- The application will collect a set of hardware processing elements that will be used to execute all computation and SDA threads, and on which to store all data structures.
- Each computation task will be represented by a group of computation threads, distributed over some set of processors. Any data structures belonging to the task will also be distributed over the same set of processors, according to their respective distribution directives.
- Each SDA will be distributed over some set of processors, where the presence of an SDA on any given processor is evidenced by some set of distributed data and an SDA thread responsible for that data.
- Each thread group will identify a "master" thread that can act as a negotiator for the entire thread group on issues such as determining data distribution.
- Multiple computation and SDA threads on the same processor will be executed in an interleaved fashion, according to the scheduling policy of the thread system and the "readiness" of the threads. Priorities can be used to influence thread scheduling decisions. For example, when a message arrives for an SDA thread, it may assume higher priority than the other computation threads, allowing it to handle the message at the next scheduling opportunity.

This level of complexity in data structure management is necessary to accommodate the various modules which comprise an MDO application, since each module is typically developed independently of the others and wishes to 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.

3.2 SDA Method Invocation

Referring once again to Figure 5, we now discuss the runtime issues involved with 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 object 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. In the previous section we addressed with the issues arising due to distributed SDA data structures. We now address the issues regarding SDA control structure and method invocation.

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. Allowing for distributed control of an SDA would require implementing distributed mutual exclusion algorithms, such as [21], to guarantee the monitor-like semantics of SDAs, and is a point of interest for future research.

Mutually-exclusive access to the SDA in the current design, is guaranteed by the fact that each SDA is controlled by a single *master* thread. When an SDA method is invoked by a thread executing on a different processor a message is sent to the processor executing the SDA master thread with a request to invoke the method. The calling task then waits for the reply. Since the SDA master is located on a single processor, and all actual method invocations are performed by the master, we maintain the monitor-like semantics of SDA method invocation.

Having established the master-worker organization of the SDA control structure, we can now describe a simple mechanism for ensuring that the second restriction is enforced. Each SDA method has an associated boolean condition function representing the condition expression specified by the programmer. When an SDA master 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. When no more enqueued method conditions evaluate to true, a new method invocation request is processed. Starvation is prevented by ensuring that any enqueued method whose condition has changed, is processed before a new method request. Fairness is determined by the order in which enqueued method conditions are evaluated, which is under the programmer's control.

4 A Prototype Implementation

We now describe a prototype implementation of the SDA runtime system for method invocation. At this point, all data structures are allocated to a single processor, and the SDA control structure is centralized. The prototype is built using C++ and is currently running on a cluster of workstations, supported by the **p4** [4] portable primitives for parallel execution. To test the feasibility of our method invocation design, we have successfully implemented and executed two SDA programs using this prototype: a simple stack manipulation program and the MDO application for aircraft design introduced in Section 2.1.

To explain the details of our prototype implementation, we present the SDA *stack* example, which provides several public methods, such as *push*, *pop*, and *top*. Each method is guarded by a condition clause to ensure that the stack does not overflow or underflow. For example, the *pop* and *top* methods may only be invoked when the stack is non-empty, while the *push* may only be executed so long as the stack is not at its maximum size. If a task thread sends a request to *pop* an empty stack, the condition clause will evaluate to false and the method will be enqueued until the condition clause can be satisfied. When (perhaps at a later time) another task thread sends a *push* request to the same stack and, assuming the stack is not full, its condition clause will evaluate to true, the *push* will be executed, and an acknowledgment returned to the caller. The condition for the blocked *pop* will now evaluate to true since the stack is now non-empty, and the resulting value returned to its caller in an acknowledgment message.

Each SDA method is compiled into three functions:

1. the *method* function, which embodies the method code itself as specified by the programmer,

```
TYPE SDA-stack-pop-stub ()
{
        send-message-to-SDA (POP)
        receive-message-from-SDA (result)
        return(result)
}

bool SDA-stack-pop-condition ()
{
        return (stack-height > 0)
}

generic SDA-stack-pop-method()
{
        result = pop-from-private-stack ()
        return (result)
}
```

Figure 8: Three functions needed to implement the SDA stack method *pop*

2. the *condition* function, which is a boolean function that evaluates the guarded condition clause, and
3. the *stub* function, which provides the method's public interface to the task threads and is used to access the SDA *method* function from a remote processor.

Figure 8 depicts the pseudocode for the SDA stack method *pop*. When a task makes a method call, it is actually invoking the *stub* routine for the specified method, which marshals the arguments into a buffer, sends a generic message to the SDA master, and awaits a reply. Each message consists of (1) the name of the method to be invoked, (2) the callers address (used for the reply message), and (3) the method arguments.

Each SDA master is a thread which waits for messages from task stubs and takes appropriate action as specified by the message. The master incorporates a data structure analogous to that shown in Figure 9 for the *stack* SDA. The data structure consists of a list of queues, one for each method, and associated with each method queue are pointers to its *condition* and *method* functions. Both functions are called using the single generic argument pointer that was created by the *stub* code and sent with the method invocation request. The *condition* function always returns a boolean, whereas the *method* function returns a generic value that is sent back to the *stub* function. Along with pointers to the *method* and *condition* functions, each method queue also contains a list of outstanding method invocation requests, represented by the message received from the *stub* routine.

The algorithm in Figure 10 depicts the main loop of the SDA master. On receiving a message from a task *stub* routine, the SDA master executes the associated *condition* function to determine if the method can be executed. If the *condition* function returns false, the method is enqueued in the appropriate list. Otherwise, the associated *method* function is executed and the results returned to the caller through the *stub* routine. Since the execution of any method may change the SDA state, the *condition* functions associated with any enqueued SDA methods are reevaluated and the methods whose conditions evaluate to true are executed. This reevaluation of *condition* functions is repeated until no further methods can be executed, at which time the master continues waiting for further messages from *stub* routines.

4.1 Preliminary Results

In order to quantify the overhead of our method invocation design, we performed the following experiment:

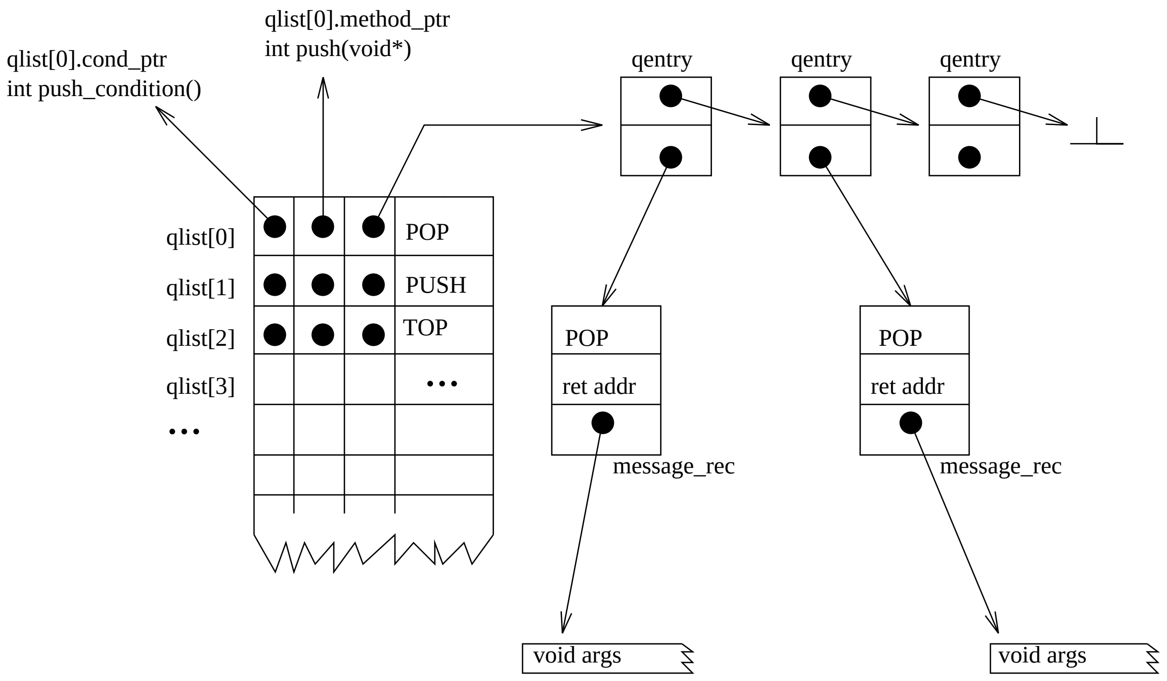


Figure 9: The queue list data structure used in the *stack* SDA prototype

```
sda-master()
{
   do forever
   {
      m = wait-for-message ();
      if (m.condition (m.arg) == true)
      {
         result = m.method (m.arg);
         return-result-to-caller (result, m.caller);
         repeat
         {
            for each method queue do
            {
               while condition of queue i is true
               {
                  m = get-first-method-in-queue (i)
                  result = m.method (m.arg)
                  return-result-to-caller (result, m.caller)
               }
            }
         } until no methods are executed in a single pass
      }
      else
        enqueue-message (m);
   }
}
```

Figure 10: Pseudocode for the main loop of an SDA master

1. We first measured the round-trip message delay of our network using p4. This provides the basis for comparing our other results, since in all cases a single message is passed from the calling stub to a master thread, and back to the calling stub.

2. Next, we encoded a version of our SDA stack routines using a simple remote procedure call mechanism, but without the complicated SDA master algorithm and queue structures. This is done by performing a sequence of pushes and pops without checking for underflow/overflow conditions. Since our SDA method must do at least this much work, this test represents a lower bound to the execution of our SDA stack.

3. Next, we timed our prototype SDA implementation performing a series of alternating pushes and pops. However, although the conditions are evaluated for each method, they always evaluate to true and so we never enqueue any methods. This test is designed to highlight the overhead associated with packing and unpacking messages from the stubs and evaluating condition routines.

4. Finally, we timed our SDA stack prototype using a series of alternating pushes and pops, but this time we simulate each method function evaluating false on the first time, true on the second. This forces the SDA master to enqueue the arriving method, then retrieve the method, re-evaluate its condition function, and finally complete its execution. This test is designed to measure the overhead associated with manipulating the method queues.

Each of the four tests consisted of multiple trials of 10,000 complete method calls, and each trial was timed (including message delays) to determine the average time required for a method call (or a round-trip message in the case of test #1). The different trials were then averaged together to achieve confidence intervals of 95%. All tests were performed under similar conditions on a set of Sun Sparcstation 10 workstations with low traffic.

The outcome of the tests, presented in Figure 4.1, yield the following empirical results:

- the basic RPC mechanism adds only 1.7% overhead to the basic round-trip message time (test #1 vs. test #2), where the relative overhead of test #2 to test #1 is computed as: $(t_{test\#2} - t_{test\#1})/t_{test\#1}$. Remaining overheads are computed similarly.

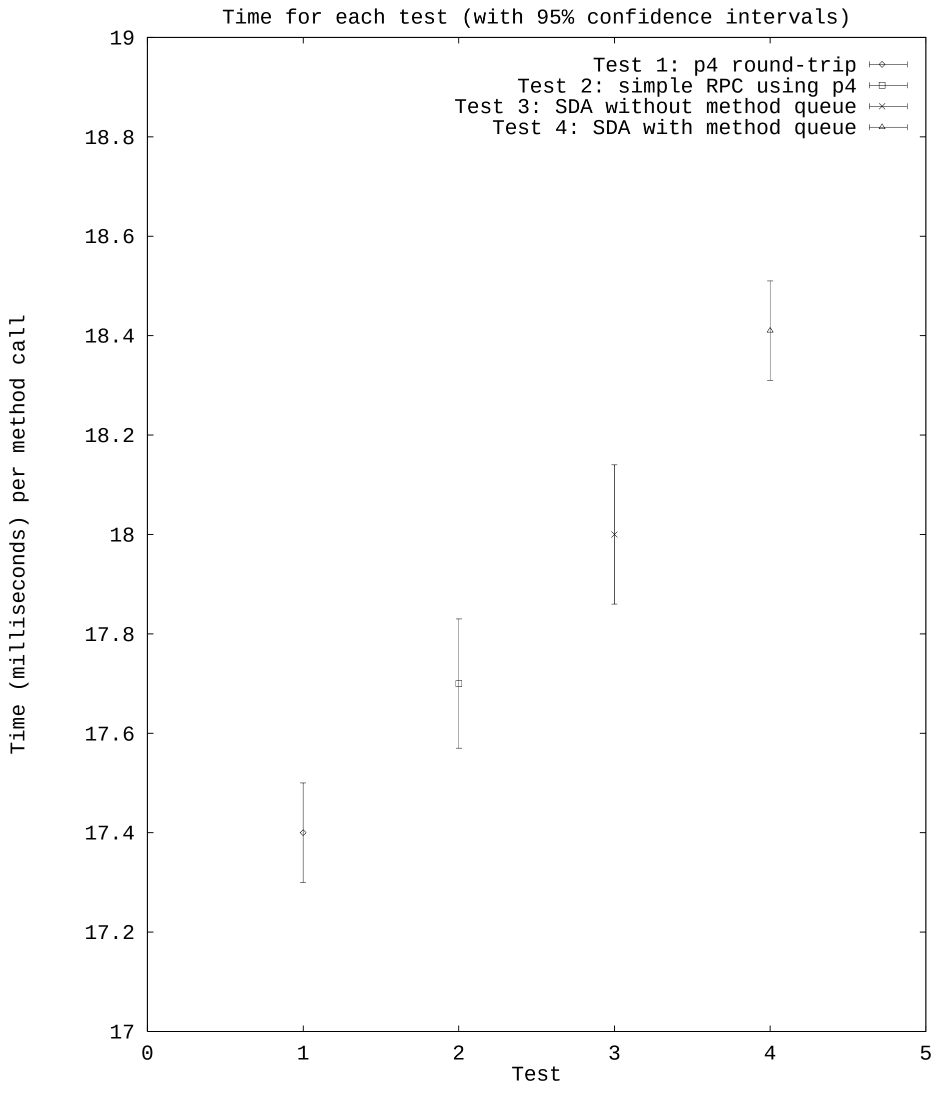
- the overhead for marshaling messages and executing condition functions adds only 1.7% overhead to the basic RPC mechanism (test #3 vs. test #2), which is 3.4% overhead as compared with the basic round-trip message time (test #3 vs. test #1);

- the overhead for manipulating the method queue structures is 2.3% (test #4 vs. test #3), so the overhead for the entire SDA method invocation mechanism adds only 4.0% overhead to the basic RPC mechanism (test #4 vs. test #2), which is only 5.8% overhead when compared with round-trip message time (test #4 vs. test #1).

We acknowledge that some overheads may be obscured by the large round-trip message latency of the p4 package, and that a leaner message passing interface may yield higher overheads, and we are currently in the process of porting our prototype to other platforms, such as the Paragon. We plan to report on these results in the final version of this paper.

5 Conclusions and Future Research

We have introduced a mechanism for integrating task and data parallelism by extending the HPF definition with a set of primitives for defining and controlling parallel tasks and shared data abstractions (SDAs). We focus on the runtime support necessary to support such a system, and provide design details for the two


Time for each test (with 95% confidence intervals)
Test 1: p4 round-trip
Test 2: simple RPC using p4
Test 3: SDA without method queue
Test 4: SDA with method queue
Time (milliseconds) per method call
Test
17
17.2
17.4
17.6
17.8
18
18.2
18.4
18.6
18.8
19
0
1
2
3
4
5

segments of the runtime support: distributed data structures and method invocation using monitor and guarded condition semantics.

In particular, we describe the implementation of our method invocation design as a C++ prototype capable of executing SDA and task codes with centralized SDA control and data structures. Using this prototype we have implemented and executed two systems employing task parallelism and SDAs: a simple stack program which performs randomized pushes and pops, and a multidisciplinary optimization (MDO) application for aircraft design. Both codes execute on a cluster of workstations using p4 as the communication library.

Using our prototype implementation, we have measured the expected overhead of our method invocation design and have found that our design adds little overhead to a stripped-down RPC version of the same test, which we feel is a realistic lower bound for remote method invocation. We plan to expand our results on the overheads of method invocation for the final version of the paper.

We are currently in the process of incorporating the Chant runtime system into our prototype, which will give us the capability to exercise our distributed SDA data structure designs, as well as full support of our parallel tasks and portability to a large number of platforms, including the Paragon and KSR-1. From the experiences and results of our prototype, we are making modifications to the SDA syntax and semantics, and are working on a source-to-source translator that will take HPF programs augmented with our SDA syntax and produce code that will execute on distributed memory multiprocessors and workstation clusters. Finally, we are working with several applications groups to develop realistic MDO codes that will provide the true test of our designs.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Symposium on Operating Systems Principles*, pages 95–109, 1991.

[2] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Scalable High Performance Computing Conference*, pages 51–59, Williamsburg, VA, April 1992.

[3] 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.

[4] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Matthew Haines and Wim Böhm. On the design of distributed memory Sisal. *Journal of Programming Languages*, 1:209–240, 1993.

[16] 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.

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

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

[19] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[20] Jenq Kuen Lee and Dennis Gannon. Object oriented parallel programming experiments and results. In *Proceedings of Supercomputing 91*, pages 273–282, Albuquerque, NM, November 1991.

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

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

[23] 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.

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

[25] Carl Schmidtmann, Michael Tao, and Steven Watt. Design and implmentation of a multithreaded Xlib. In *Winter USENIX*, pages 193–203, San Diego, CA, January 1993.

[26] H. Schwetman. *CSIM Reference Manual (Revision 9)*. Microelectronics and Computer Technology Corperation, 9430 Research Blvd, Austin, TX, 1986.

[27] 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.

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

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

[30] Sun Microsystems, Inc. *Lightweight Process Library*, sun release 4.1 edition, January 1990.

[31] 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.

[32] 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.