Approaches for Integrating Task and Data Parallelism

Henri E. Bal
Dept. of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
bal@cs.vu.nl

Matthew Haines
Department of Computer Science
University of Wyoming
Laramie, WY 82071-3682

USA
haines@cs.uwyo.edu

Abstract

Languages that support both task and data parallelism are highly general and can exploit both forms of parallelism within a single application. However, integrating the two forms of parallelism cleanly and within a coherent programming model is difficult. This paper describes four languages (Fx, Opus, Orca, and Braid) that try to achieve such an integration and identifies several problems. The main problems are how to support both SPMD and MIMD style programs, how to organize the address space of a parallel program, and how to design the integrated model such that it can be implemented efficiently.

Keywords: Parallel programming systems, task parallelism, data parallelism, shared objects, coordination languages, Fx, Opus, Braid, Orca, HPF.

Introduction

Most parallel programming systems are based either on task parallelism or on data parallelism. Task parallelism (also known as control or process parallelism) allows the programmer to define different types of processes. These processes communicate and synchronize with each other through message passing or other mechanisms. In between synchronization points, each process executes independently from all other processes. Task parallelism is used in many parallel languages [1].

Data parallelism is a quite different model and is based on applying the same operation in parallel on different elements of a data set. Unlike with task parallelism, all processors conceptually execute the same program, on different data elements. The advantage of data parallelism is that it uses a simpler model. As shown by High Performance Fortran (HPF) and other languages, the task of the programmer can be reduced substantially by providing the right language and compiler support. In HPF, the programmer mainly is responsible for specifying the distribution of data structures. The compiler takes care of generating the necessary code for communication and synchronization. With a task parallel programming system, on the other hand, the programmer must deal explicitly with process creation, communication, and synchronization.

Unfortunately, the simple model of data parallelism also makes it less suitable for applications

that do not fit the model. In particular, applications that use irregular data structures often do not match the model and impose difficult problems on both the language designer and compiler writer. Such applications are often easier to write in a task parallel language.

Since both task and data parallelism thus have their strengths and weaknesses, it is attractive to integrate both forms in one model. Recently, several languages have been proposed that try to achieve such integration [4, 6, 8, 12, 16, 17, 19]. Systems like Fx, Opus, and Fortran-M add task-parallel constructs to HPF (or other data-parallel Fortran dialects), with the goal of making HPF more general-purpose. Other research projects, such as Orca and Braid, are working in the opposite direction, and add data-parallel constructs to a task-parallel language, primarily to take advantage of the ease of data parallel programming.

In this paper, we will survey the approaches for integrating task and data parallelism. We will first look in more detail at the reasons for integrating the two models and at applications that can benefit from the integration. Next, we discuss why the integration is difficult. As we will see, many languages succeed in obtaining a certain degree of integration, but no language has all advantages of both models. Often, a language is biased and favors either data parallelism or task parallelism. We will describe a number of problems and use these for studying four proposed languages in some detail.

Integrating task and data parallelism

Below, we discuss the issues of why integrating task and data parallelism is a worthwhile pursuit. We have identified at least three advantages for integrating task and data parallelism within the same language. The first advantage is *generality*. Given the large number of parallel programming models and languages available, having a single language that is capable of encoding a wide-variety of parallel applications is certainly attractive.

The second advantage of integrating task and data parallelism is the capability of increasing scalability by *exploiting both forms of parallelism* within a single application. For many applications, data sizes are fixed and cannot easily be increased, thereby limiting the amount of data parallelism that can be exploited. Consider the problem of processing sensor inputs, such as the Narrow-band Tracking Radar problem [7]. This problem reads a set of input matrices corresponding to sensor channel readings and performs a series of transformations on them, including Fast Fourier Transform (FFT). Although some of the transformations can be done in a data parallel manner, the amount of parallelism that can be exploited is limited by the size of the matrices produced by sensor channels. However, since a new matrix is produced at each time interval, additional parallelism can be exploited by implementing a set of the transformations as parallel tasks. See the sidebar on applications for more details on the Narrow-band Tracking Radar problem.

The third advantage of integrating task and data parallelism is to enable the *coordination of multidisciplinary applications*. Many modern scientific applications are created as a collection of subprograms from a variety of different disciplines that are integrated into a single, multidisciplinary application. For example, consider the design of a modern aircraft, which involves the coordination of several discipline models including aerodynamics, propulsion, and structural analysis. Each model is typically encoded to execute in data parallel. Combining these independently-designed computer models, or discipline codes, into a single application requires complex synchronization and communication between the individual models. Therefore, to efficiently coordinate the execution of these independent data parallel models, which can be viewed as very large-grained tasks, a language that supports both task and data parallelism is required. See the sidebar on applications for more details on a multidisciplinary application for aircraft design.

Problems with integrating task and data parallelism

As should be clear now, there are many advantages to having a single, integrated programming model that supports both task and data parallelism. Unfortunately, integrating the two models is far from easy, as they have many differences. Below, we look at the most important problems in the integration. The first two problems are due to the way parallelism and communication are expressed in the two models, whereas the third problem is related to the implementation.

A key difference between task and data parallel languages is the way a parallel program is structured. With data parallel languages, there is a single program that is executed on all machines, resulting in a Single-Program Multiple Data (SPMD) model. In contrast, task parallel programs typically consist of many different types of processes that execute largely independently from each other. Such programs are written in an MIMD (Multiple Instructions Multiple Data) style. An integrated model should support both styles (SPMD and MIMD), but clearly they are quite different. As we will see, most languages we discuss are biased either to an SPMD or MIMD style.

A second key difference between task and data parallel languages concerns the organization of the address space. In modern data parallel languages (such as HPF, Fortran-D, and many others), parallelism occurs in a single, global, shared address space. In a distributed implementation, the compiler takes care of generating code for data transfers, transparently to the user. Most task parallel languages, on the other hand, provide a separate address space for each process, and require the programmer to insert explicit send and receive statements to transfer data between these disjunct address spaces. Again, this difference makes a smooth integration of task and data parallelism difficult.

Not all task parallel languages use multiple address spaces. Distributed Shared Memory (DSM) systems provide a logically shared address space on top of a distributed-memory architecture. DSM-

based systems thus may be easier to integrate with data parallelism than message passing systems. One example is Shared Virtual Memory (SVM) [13], which simulates physical shared memory using page-based techniques. Integrating SVM with a data parallel system will be far from straight forward, however, because SVM partitions the address space into fixed-size pages, which may not match the partitioning required by the application. Languages like HPF therefore allow the user to specify the distribution of data, using knowledge about the application. Another DSM-based system is Linda, which provides an associative shared memory called Tuple Space. This Tuple Space can be used as a basis for integrating task and data parallelism [3]. Later in this paper, we will look at yet another form of DSM, based on shared objects, and see how suitable it is for integrating task and data parallelism.

A third difference between task and data parallel languages is the way they are implemented. Data parallel languages like HPF rely on extensive compiler analysis, for example to compute efficient communication schedules or to vectorize multiple data transfers using a single message [2]. Consequently, these languages are designed to allow such analysis to be done statically. In particular, the data distribution is usually known at compile-time.

Task parallel languages use much less heroic compilers. In fact, many of these languages use traditional compiler technology designed for sequential languages. Unlike data parallel languages, however, they often use complicated runtime systems. Even traditional message passing languages usually have extensive runtime systems that buffer, order, and filter incoming messages. DSM-based systems use even more complicated techniques, such as data replication.

Since task parallel languages do not make high demands on the compiler, they often have fewer restrictions than data parallel languages. Typically, they allow the dynamic creation of processes and allocation to processors. If such task parallel constructs are added to a data parallel language, it will make compile-time analysis much more difficult, if not impossible. To prevent this, languages like Fx impose restrictions on the task parallel constructs and require processor allocation to be specified statically. An interesting alternative would be to use run-time compilation [14], but this technique is exploited in few systems yet.

Approaches to integrating task and data parallelism

We describe four systems that integrate task and data parallelism in different ways: Fx, Opus, Orca, and Braid. The goal of this work is to determine to what extent these languages succeed in integrating task and data parallelism. We recognize that there are other systems which integrate task and data parallelism (see the Sidebar on Related Work), but this is not intended to be an extensive survey of the field. Rather, we selected two languages with a task parallel background and two languages with a data parallel background.

Fx

Programming model

Fx [10, 16] adds task parallel directives to a data parallel language based on HPF. A *task* corresponds to an execution of a *task-subroutine*, which is a data parallel subroutine with well-defined side-effects: only the actual arguments to the subroutines may be modified. Contrary to most task-parallel languages, all tasks in Fx share a common name space, and all communication between tasks occurs at procedure boundaries and is generated by the compiler rather than being specified explicitly by the programmer. Task parallelism is achieved through the execution of *parallel sections*, which are defined as a collection of task-subroutine invocations. Within a parallel section, all tasks (subroutines) may execute in parallel.

Execution of an Fx program begins as a single data parallel program running on all of the statically-allocated processors. When a parallel section is encountered, all task-subroutine invocations within the section are executed in parallel, provided none of the data dependence constraints are violated. Since all interaction between tasks occurs at procedure boundaries, the data dependence constraints are specified by the subroutine arguments. Once all of the task-subroutines have completed (i.e., barrier), they exit and Fx resumes data parallel execution on all processors.

Implementation

Fx is a prototype compiler whose current targets include an iWarp parallel machine, networks of workstations running PVM, and the Cray T3D. The Fx compiler generates code for both allocating tasks to processors and for communicating between the tasks during task-parallel execution. To do this, the compiler relies on two forms of directives from the programmer: argument input/output and mapping. These directives are specified by the programmer for each specification of a task-subroutine.

The *argument input/output directives* (**input**, **output**) define the scope of the data space that a task-subroutine accesses and modifies. Every variable whose value may be used by a task must be specified with one of these scoping directives. These variables may be scalars, arrays, or valid Fortran 90 array sections with the restriction that all bounds and step sizes must be statically known (i.e., constant).

The *mapping directives* (**processor**, **origin**) specify how a task is to be mapped onto a statically-allocated set of processors. **Processor** declares how many processors are to be assigned to a given task-subroutine, and **origin** indicates the location of the task-subroutine within the parallel system. Both processor and origin directives assume a two-dimensional processor space with origin (0,0) at the top-left of the mesh. Figure 1 (taken directly from [10]) shows the input/output and mapping directives, along with an illustration as to how the proposed mapping would appear on an 8x8

```
c$  begin parallel
    do i = 1, n_iters
      call colffts (A)
c$      output (A)
c$      processor (8,3)
c$      origin (0,0)
      call rowffts (A)
c$      input (A), output (A)
c$      processor (2,5)
c$      origin (0,3), (2,3), (4,3), (6,3)
      call hist (A)
c$      input (A)
c$      processor (2,5)
c$      origin (0,3), (2,3), (4,3), (6,3)
    enddo
c$  end parallel
```

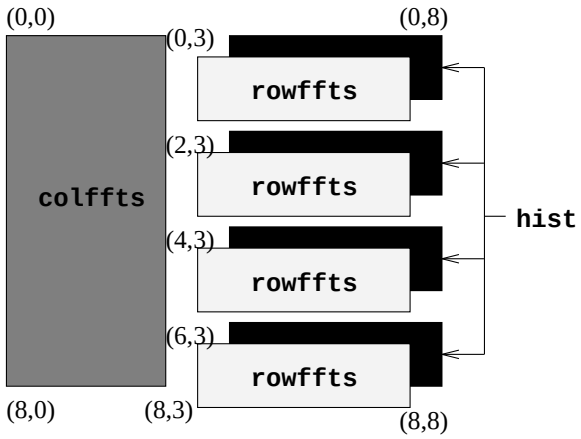

Figure 1: Fx code segment and corresponding mappings on an 8x8 parallel machine.

parallel machine. Two parallel sections assigned to the same processor set, such as `rowffts` and `hist` in Figure 1, are scheduled in round-robin fashion.

The Fx compiler generates a task-parallel program from the I/O and mapping directives by using the task-subroutines to identify the tasks and the parallel sections to identify those tasks that are to execute concurrently. The allocation of tasks to processors is extracted from the mapping directives. Dependencies between the tasks are identified by combining static flow analysis with the argument input/output directives. For example, given the code in Figure 1, the compiler would identify the dependency on matrix A between the three task-subroutines. Finally, communication is generated according to the allocation and dependencies of the tasks; if a dependency exists between two task-subroutines and they are allocated to different processor sets, then communication primitives are generated to exchange the necessary information. Since task-subroutines can only affect their arguments, all communication is limited to subroutine boundaries.

Discussion

Fx achieves a clean integration of task and data parallelism that is easy to program. This is because much of the work of traditional task-parallel programming, such as mapping, communication, and synchronization, is done with simple directives to the compiler. In Fx, task parallel and data parallel programs are always identical except for directives and therefore the results are always consistent with sequential and data parallel execution. However, the tradeoff for this ease-of-use is a restricted form of task parallelism in which 1) communication can only occur indirectly through subroutine arguments at procedure boundaries, and 2) mapping is fixed at compile-time. In addition to having to re-compile a program when input sizes are changed, it may also be necessary to adjust the mapping

directives since the best mapping depends on the input size.

In conclusion, Fx is well-suited for scientific applications that want to use task parallelism to increase scalability. However, Fx is not sufficient for dynamic programming problems in which traditional concurrent programming is used, or for coordinating the execution of multiple data-parallel applications. In both cases, Fx lacks general communication and synchronization capabilities.

Opus

Programming model

Opus [4] adds task parallel programming constructs to HPF for the purpose of coordinating concurrent execution of several data-parallel components. Therefore, a *task* in Opus is defined as the execution instance of a data parallel program. In this regard, Opus tasks are very coarse-grained as compared with the other languages described in this paper.

Opus introduces a programming component called a *ShareD Abstraction*, or SDA, to support interaction between concurrently executing tasks in a safe, high-level manner. Like most Abstract Data Types (ADT), an SDA contains a set of data structures that define its state, and a set of methods for manipulating this state. However, SDAs can be used in one of two modes: 1) as a traditional ADT that acts as a data server between two concurrently-executing tasks, or 2) as a computation server driven by a main, controlling task. As a data server, the SDA data structures define the data to be shared between tasks, and the methods define functions for accessing and manipulating this data. In this role, the SDA serves as the "glue" that combines several independent tasks, providing for both communication and synchronization. As a computation server, the SDA provides a common framework for accessing functional units and for sharing data between these units. In this role, the SDA behaves more like an Object Request Broker (ORB) [5]; it places a "wrapper" around the functional units for a given application so that they have a common interface. For example, a modeling application might be wrapped with interfaces for initializing the model, performing one step of the model, and cleaning up. These common interfaces then become "buttons" that can be manipulated by a central coordinating task.

Execution of an Opus program begins with a single coordinating task to establish all of the participating computation servers and data servers. The coordinating task then "drives" the computation by invoking the proper methods within the computation SDAs. Communication and synchronization between the concurrently-executing computations is managed by the data SDAs. For example, if one computation creates an array and another performs some operation on that array, then the SDA controls passing the array from one computation to the other, including any possible remapping that may be needed, as well as synchronizing the two computations based on their shared data dependence.

Implementation

Although the syntax and semantics of the Opus language have been completely specified, an actual compiler has not been fully implemented. However, a prototype runtime system that supports SDAs as data servers does exist [11]. In order to support concurrent execution, it must be possible for SDA methods to execute independently of the invoking task. This is accomplished by using a separate thread of execution that is mapped onto a set of processors as specified in the declaration of an SDA. This represents a *non-blocking* invocation of an SDA method, where the caller is free to continue work after the invocation without having to wait for its completion. This decision of whether an SDA invocation should be blocking (sequential) or nonblocking (concurrent) is supported at the call site so that the same method may be invoked in whichever manner appropriate.

As a data server, the SDA data structures will be shared by several cooperating tasks executing concurrently. Figure 2, for example, illustrates a simple data server that implements a FIFO bounded buffer. Also, data parallel access to SDA data structures is supported through distribution of the internal data structures using HPF syntax. Therefore, method invocation must address synchronization concerns as well as data (re)mapping and communication. To control synchronization and honor data dependencies, each method declaration can have an optional *when* clause which "guards" the execution of the method until the specified logical expression can be satisfied. Logical expressions are limited to comparing the internal state of an SDA and the method arguments (if any). For example, in Figure 2 the `put` and `get` methods are guarded by logical expressions to check the size of the buffer. To handle data distribution within an SDA, method arguments that are passed from the computation processor(s) to the SDA processor(s) may need to be redistributed. This is accomplished through a handshaking protocol that occurs between the invoking computation and the SDA, in which the distribution information about the actual arguments (source) and the formal arguments (destination) are exchanged. Based on these distributions, a remapping table is computed, generating a communication schedule that is executed to perform the actual data transfer.

Discussion

Opus is designed to allow concurrent execution of several independent, data-parallel tasks that interact with one another through well-defined distributed data abstractions. This is the perfect situation for coordinating the execution of a multidisciplinary optimization (MDO) application, in which several independently-generated models are combined to form a single application. However, doing so requires a complicated (and expensive) method invocation overhead for these data abstractions, and so the size of these cooperating tasks must be large enough to amortize this cost. Therefore, Opus is not well-suited as a programming language for traditional find-grained task-parallel applications.

```
SDA TYPE  buffer_type (size)
   INTEGER :: size
   REAL, PRIVATE :: fifo(0:size-1)
   INTEGER, READ_ONLY :: count=0
   INTEGER, PRIVATE :: px=0          ! producer index
   INTEGER, PRIVATE :: cx=0          ! consumer index
         ...
CONTAINS
   SUBROUTINE put(x) WHEN (count .LT. size)
      REAL, INTENT(IN) ::  x
      fifo(px) = x
      px = MOD(px+1,size)
      count = count + 1
   END

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

Figure 2: Opus SDA data server implementing a bounded buffer.

Data Parallel Orca

Programming model

The Orca task parallel language has been extended with constructs for data parallelism, resulting in a language with mixed parallelism [12]. Orca supports general task parallelism and allows dynamic creation of processes and mapping to processors. Communication in Orca is not based on message passing but on *shared objects*, which are instances of Abstract Data Types (ADTs). Processes in Orca communicate by applying user-defined ADT operations on shared objects. Such an operation can be applied to only a single object and is always executed indivisibly.

In the extended Orca language, data parallelism is expressed through *partitioned objects*, which are objects containing arrays that can be distributed among multiple processors. A data parallel operation on a partitioned object is executed in parallel by these processors; each processor applies the operation to the elements it "owns" (the owner-computes rule). The distribution of the elements is expressed by the user, by invoking a runtime primitive that specifies the set of processors to use and the owner of each element. This distribution may be changed during runtime. The programmer can also cluster elements that are accessed together into so-called *partitions*. Whenever an operation needs an element from a remote processor, it will actually fetch the entire partition the element

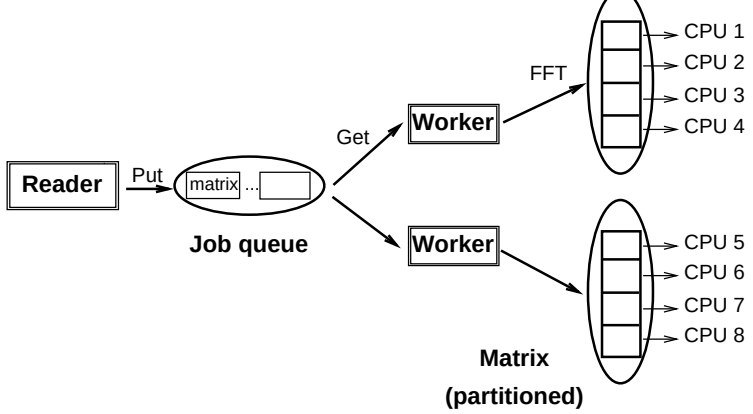

Figure 3: Structure of the Narrowband Tracking Radar problem in Orca.

belongs to, thus increasing the granularity of the data transfers (much like message vectorization).

Task and data parallelism can be used together in a single program. For example, Figure 3 shows the structure of the tracking radar program in Orca; Figure 4 shows some fragments of this program. The program uses master/worker task parallelism to distribute the matrices and data parallelism to parallelize the transformations. Each worker is assigned a list of processors that it can use for its data parallel FFT operations. The master puts the matrices to be transformed in a job-queue object. Each worker repeatedly gets the next matrix to work on from this object and stores it in a partitioned object that is distributed among the processors assigned to the worker. Next, the worker invokes a data parallel FFT operation on this partitioned object.

Implementation

Data parallel Orca has been implemented on several platforms (multicomputers and collections of workstations), by extending the Orca compiler and runtime system. The main implementation issue is how to implement shared objects. Objects that are not partitioned are stored either on one machine or on all machines that can access the object. Single-copy objects are accessed using remote object invocations. For replicated objects, read-only operations are executed using the local copy, without doing communication; write operations are executed by updating all copies, using totally-ordered broadcasting to send the operation to all machines.

A data parallel operation on a partitioned object is executed by broadcasting the operation; each processor then executes the operation on the elements it owns. If a processor needs to read values stored on another machine, these values are either fetched on demand or they are prefetched before the operation begins. Prefetching is only possible if the compiler (or the programmer) conveys the data dependencies of the operation to the runtime system. Except for this data dependency analysis, the system uses little compiler support. The implementation is mainly based on an extensive runtime system.

```
PROCESS worker(Q: SHARED JobQueue; Procs: CPUListType; NProcs: integer);
  M: ARRAY[1..N,1..N] OF complex;
  Matrix: MatrixObject[1..N,1..N];     # a partitioned object
BEGIN
  Matrix$$rowwise_partitioning();
  Matrix$$distribute_on_list(Procs, NProcs, BLOCK, ...);
  WHILE Q$Get(M) DO   # retrieve a matrix from the queue
    Matrix$init(M);   # use it to initialize the object
    Matrix$FFT();   # invoke FFT
  OD;
END;

PROCESS Reader();
  Q: JobQueue;
BEGIN
  ...
  FORK worker(Q,[0,1,2,3],4) ON 0;
  FORK worker(Q,[4,5,6,7],4) ON 4;
  ...
END
```

Figure 4: Data Parallel Orca code for the Narrowband Tracking Radar.

Discussion

An important advantage of this approach is that it integrates task and data parallelism in a clean and simple way, using a general object model that supports both replicated and partitioned objects. In particular, the usage of objects (instead of message passing) avoids the problem with multiple address spaces described earlier. Orca can in fact be regarded as an object-based form of Distributed Shared Memory.

The model, however, is biased to task parallelism. It supports general task parallelism (e.g., dynamic process creation), but it imposes several restrictions on data parallelism. Programs are required to adhere to the owner-computes rule, which may not always be appropriate. Also, data parallel operations that use multiple arrays are not well supported by the model, because operations are always applied to a single object (both in the original Orca language and in the extended language). It is possible to store multiple arrays in a single object, but then they automatically are distributed in the same way, which is undesirable for applications like matrix multiplication.

Both Orca objects and Opus SDAs represent abstract data types that can be distributed over a set of processors using conventional data parallel mapping directives. Both apply operations to their elements using the owner-computes rule. Aside from implementation issues, the main difference between ADTs and SDAs is in the "server" nature of the SDA. All SDAs run implicit server loops to handle incoming requests; resulting SDA methods can be invoked either synchronously

```
dataparallel mentat class matrix{
      float value;  // the elements
   public:
      float AGG dot_product ROW (COL 1xN matrix& B);
}
float AGG matrix::dot_product ROW (COL 1xN matrix& B) {
   float result = 0.0;
   for (int j = 0; j < this.num_cols(); j++)
      result += this[j].value * B[j].value;
   return (result);
}
```

Figure 5: An example data parallel Mentat class in Braid.

or asynchronously, where the decision can be made at the call site. This allows SDAs to behave as computation servers as well as data servers. Orca objects deliberately lack such a server, thus allowing concurrent read operations on different copies of an object.

In conclusion, the extended Orca language gives the full power of task parallelism with some of the functionality of data parallelism, integrated in a clean model. It does not have the full power of data parallel languages like HPF. The language thus is biased to task parallelism, which also is reflected in the usage of extensive runtime support and little compiler support.

Braid

Programming model

Braid [19] is a data parallel extension to the Mentat Programming Language (MPL). MPL is an object-oriented task parallel language based on C++. The MPL programmer can designate certain classes as *Mentat classes*. Operations on objects of Mentat classes are executed in parallel, using a macro dataflow model. For efficiency, only classes whose operations are compute-intensive should be annotated in this way.

Braid logically extends this model by introducing *data parallel Mentat classes*. Objects of such classes are partitioned among multiple processors. Operations on these objects are executed in a data parallel way, much as in Orca.

A novel idea in Braid is *subset level data parallelism*, which can be used to define operations on entire subsets of an object's data. The example of Figure 5 (taken from [19]) illustrates this idea. A data parallel class `matrix` is defined with an *aggregate* operation `dot_product`. The `ROW` annotation in this operation specifies that the compiler should generate code to apply the operation to all rows of the matrix; the implementation of the operation contains code to iterate within a single row of the matrix.

Braid allows operations to take data parallel objects as parameters. This important feature is

also illustrated in Figure 5: `dot_product` uses a parameter `B` that is a data parallel matrix object. In this way, an operation can access multiple objects, although the parameter objects can only be read and not written.

The distribution strategy for an object is determined by the compiler and runtime system, using annotations provided by the programmer. These annotations describe the communication behavior of objects. The simplest annotation expresses the local behavior within one object. It can indicate, for example, that an operation on an element usually also accesses the four neighbors of the element. Another annotation specifies nonlocal communication behavior between objects. As an example, the annotation `(COL 1xN)` in Figure 5 specifies that `dot_product` will usually combine one subset (i.e., a row) from the invoked object with an entire column of the parameter object. Other annotations exist to specify which classes of objects often interact and which operations are the dominant ones.

Implementation

Braid has not been implemented yet, but a possible implementation is described in [18]. It uses one slave process per machine, to participate in operations on all objects that are (partly) stored on that machine. A single master process takes care of object creation and distribution and also directs the communication between the slaves.

When a data parallel operation is invoked, the master first makes sure all slaves have the data required to execute the operation, using the annotations to determine which data are needed. The annotations thus are designed in such a way that data dependencies can be resolved before the operation actually begins. Once all data have been fetched, all slaves execute the operation on their local copy of the elements.

Discussion

The Braid model is closest to that of Orca, although there also are important differences. Task parallelism is based on processes in Orca and on Mentat classes in Braid. Data parallelism is expressed through data parallel operations in both languages. An advantage of Braid is that, by supporting parameter objects, it allows operations to read multiple objects. Operations like matrix multiplication thus can be expressed in a data parallel way in Braid, whereas in Orca they must be expressed using task parallelism.

Operations in Braid are executed by prefetching all required data. Orca prefetches data whenever possible and useful, but it also allows data transfers during operation execution. This difference is also reflected at the language level. Braid requires annotations to determine which data are needed by an operation. Orca generally uses compiler analysis to determine this information whenever possible, although the programmer can also provide the information, in the form of a Partition Dependency Graph [12]. The Braid system should also use the annotations to determine automatically how

	Parallelism	Address space	Implementation
Fx	SPMD bias	shared	compiler-based
Opus	coarse-grained	shared abstractions	RTS-based
Orca	MIMD bias	shared objects	RTS-based
Braid	special classes	shared objects	annotations

Table 1: Summary of the approaches.

objects are to be distributed, but the effectiveness of this approach is hard to establish without an actual implementation. In Orca, the programmer directly specifies the distribution. In comparison, the language extensions needed by Orca are simpler and also allow dynamic redistribution. Braid, on the other hand, is somewhat more expressive, because of its parameter objects.

Conclusions

Integrating task and data parallelism in a single model is useful. Not only does it result in more general programming systems, but it also allows the exploitation of both forms of parallelism within a single application, or the coordination of multidisciplinary applications. We have identified three important problems that the design of an integrated programming model has to deal with. In Table 1 we summarize how the four programming systems described in the paper address these issues.

The first problem is how to express task and data parallelism. Fx and data parallel Orca are biased to what their base languages (HPF and Orca) support, and they therefore have somewhat limited support for the alternative form of parallelism. Both languages provide a clean programming model, however, that logically extends the original model. Opus is primarily designed as a coordination language for coarse-grained tasks. Finally, Braid takes an object-oriented approach, in which both task and data parallelism are expressed using special classes (Mentat classes and data parallel Mentat classes).

The second problem is how to organize the address space of programs. Most task parallel languages use multiple address spaces (one per process), and transfer data between these address spaces through message passing. Since this is hard to integrate with data parallelism, all four languages discussed in this paper take a different approach. Fx strictly adheres to the single shared address space model of most data parallel languages; it requires user directives (and some restrictions on intertask communication) to make this possible. Opus, Orca, and Braid all use some form of shared objects (rather than explicit message passing) for communication between tasks.

The third problem is how to implement task and data parallelism in a single system. Fx adheres to the data parallel philosophy of letting the compiler do all the hard work, driven by user-supplied directives. However, language restrictions (e.g., static processor allocation and mapping) are required

to allow an efficient implementation. The Opus design uses a source-to-source translator, generating HPF programs with calls to the Opus Runtime for SDA support. Data parallel Orca also uses little compiler support coupled with an extensive runtime system to distribute, replicate, fetch, and prefetch shared data, depending on what is best for the application. Braid is designed to use annotations from the user to determine how to distribute and fetch data.

In conclusion, the programming systems discussed here take quite different approaches to the integration of task and data parallelism. Although neither of them combines the full functionality of purely task parallel and purely data parallel languages, they all are more general purpose, without unnecessarily complicating the programming model.

Acknowledgments

We are grateful to Saniya Ben Hassen, Raoul Bhoedjang, Ian Foster, Koen Langendoen, Dave O'Hallaron, Jonathan Schaeffer, and Sajpal Subhlok for providing useful comments on the paper. The work of Henri Bal is supported in part by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.). The work of Matthew Haines is supported in part by Faculty Grant-in-aid and International Travel Programs grants from the University of Wyoming, and by the National Aeronautics and Space Administration under NASA Contract No. NASA-19480, while the author was in residence at ICASE, NASA Langley Research Center.

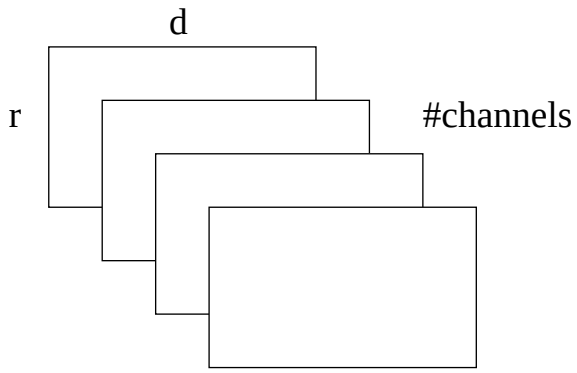

Figure 6: Input matrices for one time interval of the Narrowband Tracking Radar problem

Sidebar on Applications

To further illustrate the advantages of integrating task and data parallelism, we discuss two applications that benefit from such an integration. The first is the Narrowband Tracking Radar benchmark developed at MIT Lincoln Laboratories to measure the effectiveness of multicomputers in processing their sensor-based applications [15]. This application, and much of its description, is extracted from the CMU Task Parallel Program Suite [7], which identifies and outlines a total of five applications that benefit from an integration of both task and data parallelism.

The program is designed to receive input data from a number of channels of a given sensor. During a single time interval, each channel of the sensor produces $d = 512$ complex vectors of length $r = 10$ values. Therefore, as depicted in Figure 6, each time interval produces one matrix of size $r \times d$ for each input channel. The program then performs a number of transformations on each of the input matrices, including a *transpose* of the $r \times d$ matrix to form a $d \times r$ matrix; a *Fast Fourier Transform* of each matrix; a *reduction* in scale of each matrix; and a *threshold* cutoff evaluation of each matrix element.

The FFT, scaling, and threshold operations are typically implemented using data parallel algorithms in an attempt to improve performance. However, the amount of data parallelism is restricted by the parameters r, d, and the number of sensor channels, and cannot be arbitrarily increased to accommodate more parallelism. Rather, these parameters are a function of the sensor technology and properties of the sensor information. Therefore, to enable scalability for a large number of processors, additional task-level parallelism must be exploited. We can exploit task-level parallelism in this problem by simply assigning each input matrix set for a given time interval to a separate task, and then allowing the tasks to execute in parallel. Other mappings are also possible. By integrating both task and data parallelism, we can transform an application with limited data parallelism into one with greater task and data parallelism, thereby increasing the scalability of an application.

The second application that we will explore is a simplified multidisciplinary optimization (MDO)

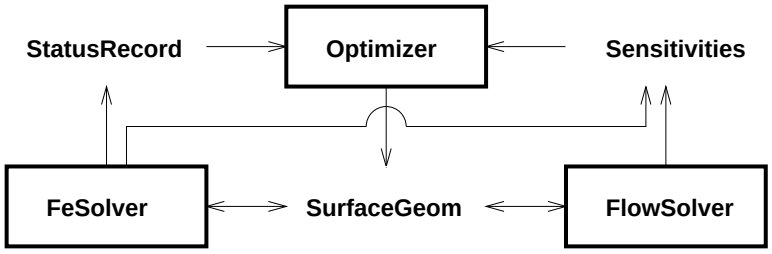

Figure 7: Interactions between independent modules from a simplified MDO application

application for aircraft design that highlights the need for supporting task parallel constructs and communication between data parallel modules. Figure 7 depicts the interactions between three independent components of a simplified MDO:

- the *Optimizer*, which initiates and monitors execution of the application until the results satisfy some objective function (such as minimal aircraft weight);
- the *FeSolver*, which generates a finite element model based on the current surface geometry to evaluate the structural integrity of the aircraft; and
- the *FlowSolver*, which generates an aerodynamics grid based on the current surface geometry and performs an analysis of the airflow around the aircraft.

While this example is greatly simplified, it illustrates a couple of key points about coordinating the execution of multiple, independent computer models. First, it is very likely that each of the independent models will exploit some form of parallelism, typically data parallelism. Therefore, executing multiple instances of these data parallel models at the same time will require an integration of task and data parallelism. Second, the interactions between the models can take different forms depending on the problem at hand and the target environment. In a sequential environment, the various models are generally executed as a pipeline. In a simple parallel variant, multiple versions of the analysis pipeline can be executed on slightly perturbed values of the design variables in order to obtain the required derivatives using finite differences. In more complex parallel versions, the models execute asynchronously, with data being exchanged at various points in the code, such as at the boundaries of the internal optimization cycles. For this latter approach, the data exchanges must be synchronized to ensure that consistency is maintained. Therefore, a language designed for the coordination of multiple models must provide proper synchronization and communication facilities.

Sidebar on Related Work

In addition to the four programming systems described in the paper, several other systems exist that also integrate task and data parallelism.

Fortran M is a set of simple extensions to Fortran 77 that support process creation and communication. Fortran M's communication model is based on message passing over *channels*. Processes and channels can be created and connected dynamically. Each channel alway has a single sender and a single receiver; this restriction was made to obtain deterministic execution semantics. Fortran M also has been used to coordinate data-parallel HPF computations.

The systems described in the paper represent language-based approaches to integrating task and data parallelism. An alternative solution is to use a library-based approach, which avoids introducing new language constructs. The HPF/MPI system of Ian Foster et al. [9] uses an HPF binding for MPI (Message Passing Interface) to implement this idea. With this system, a number of tasks can be created. Each task is written in HPF and runs (in a data parallel way) on a specified collection of machines. The tasks can communicate with each other through a subset of MPI's message passing and collective communication primitives. Several benchmarks and applications of HPF/MPI are described in [9].

Another recent approach to integrating task and data parallelism comes from the High Performance Fortran consortium. For HPF 2.0, there are three directives that have been added relating to parallel execution. The first is the `ON` directive, which partitions data parallel computations among the processors of a parallel machine. The second is the `RESIDENT` directive, which helps with the data locality problem by asserting that certain data accesses do not require interprocessor data movement for their implementation. The third is the `TASK_REGION` construct, which provides an abstract form of task parallelism. Task parallelism is expressed implicitly by mapping data objects onto processors with the `ON` directive, and the `TASK_REGION` directive allows the users to specify that disjoint processor subsets can execute blocks of code concurrently. Since HPF 2.0 is still in the definition phase, it is not yet clear how these constructs will perform, though the implementation is likely to closely resemble the compiler-based approach that Fx uses.

References

[1] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[2] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The Paradigm Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, Oct. 1995.

[3] N. Carriero and D. Gelernter. Data Parallelism and Linda. In *Lecture Notes in Computer Science 757*, pages 145–159, 1992.

[4] Barbara Chapman, Matthew Haines, Piyush Mehrotra, John Van Rosendale, and Hans Zima. Opus: A coordination language for multidisciplinary applications. *Scientific Programming*, 6(2), April 1997.

[5] The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995. http://www.omg.org/corba2/cover.htm.

[6] M. Dhagat and R. Bagrodia. Integrating Task and Data Parallelism in UC*. In *1995 International Conference on Parallel Processing, Vol. II*, pages 29–36, 1995.

[7] Peter Dinda, Thomas Gross, David O'Hallaron, Edward Segall, James Stichnoth, Jaspal Subhlok, Jon Webb, and Bwolen Yang. The CMU task parallel program suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March 1994. http://www.cs.cmu.edu/ fx/tpsuite.html.

[8] I. Foster. Task Parallelism and High-Performance Languages. *IEEE Parallel and Distributed Technology*, pages 27–36, Fall 1994.

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

[10] T. Gross, D.R. O'Hallaron, and J. Subhlok. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel and Distributed Technology*, pages 16–26, Fall 1994.

[11] Matthew Haines, Bryan Hess, Piyush Mehrotra, John van Rosendale, and Hans Zima. Runtime support for data parallel tasks. *Frontiers 1995*, pages 432–439, 1995.

[12] S. Ben Hassen and H.E. Bal. Integrating Task and Data Parallelism Using Shared Objects. In *10th ACM International Conference on Supercomputing*, pages 317–324, Philadelphia, PA, May 1996.

[13] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[14] R. Ponnusamy, Y-S. Hwang, R. Das, J.H. Saltz, A. Choudhary, and G. Fox. Supporting irregular distributions using data-parallel languages. *IEEE Parallel and Distributed Technology*, Spring 1995.

[15] G. Shaw, R. Gabel, D. Martinez, A. Rocco, S. Pohlig, J. Noonan, and K. Teitelbaum. Multiprocessors for radar signal processing. Technical Report 961, MIT Lincoln Laboratory, November 1992.

[16] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–22, San Diego, CA, May 1993.

[17] N. Sundaresan and D. Gannon. A Thread Model for Supporting Task and Data Parallelism in Object-Oriented Parallel Languages. In *1995 International Conference on Parallel Processing, Vol. II*, pages 45–49, 1995.

[18] E.A. West. Combining control and data parallelism: Data parallel extensions to the mentat programming language. Technical Report CS-94-16, University of Virginia, May 1994.

[19] E.A. West and A.S. Grimshaw. Braid: Integrating Task and Data Parallelism. In *Proc. Frontiers 95*, pages 211–219, McLean, VA, Feb 1995.