Braid: Integrating Task and Data Parallelism

Emily A. West
Andrew S. Grimshaw
Department of Computer Science, University of Virginia
{west | grimshaw @virginia.edu}

Abstract

Archetype data parallel or task parallel applications are well served by contemporary languages. However, for applications containing a balance of task and data parallelism the choice of language is less clear. While there are languages that enable both forms of parallelism, e.g., one can write data parallel programs using a task parallel language, there are few languages which *support* both. We present a set of data parallel extensions to the Mentat Programming Language (MPL) which allow us to integrate task parallelism, data parallelism, and nested task and data parallelism within a single language on top of a single run time system. The result is an object-oriented language, Braid, that supports both task and data parallelism on MIMD machines. In addition, the data parallel extensions define a language in and of itself which makes a number of contributions to the data parallel programming style. These include subset-level operations (a more general notion of element-level operations), compiler provided iteration within a data parallel data set and the ability to define complex data parallel operations.

1: Introduction

Many parallel languages have been introduced over the past decade. Broadly speaking most can be categorized as supporting either data parallelism or task parallelism (alternatively called control parallelism). Both task and data parallel languages have applications for which they are well suited. For example, task parallel languages are well suited for applications where there are multiple actions to be performed, e.g., a pipeline, or different interacting entities to be modeled. Similarly, data parallel languages are well suited to applications where the same action is to be performed on many different data items, for example image convolution where each pixel can be processed in parallel.

On the other hand, applications exist which contain a mix of the two types of parallelism. These applications force the designer to make a choice. If one type of parallelism dominates the computation, then a language to exploit that parallelism is called for and the other type can be ignored. However, for applications containing a balance of task and data parallelism the choice of language is less clear.

As an example, global climate modeling [18] involves the interaction of distinct oceanic and atmospheric weather models. The two resulting PDEs can be individually solved in a data parallel fashion by superimposing a grid upon the space to be modeled. Each grid location is then an item in a data set. Because the interactions between neighboring grid points along the borders of the oceanic and atmospheric models are quite different from the interactions between grid points internal to either model, a distinction is made between the oceanic and atmospheric portions of the computation. This calls for the use of task parallel constructs to govern inter-model interactions.

Global climate modeling is readily identifiable as an application which would benefit from a language supporting both task and data parallelism. However, one must ask if this example is unique, or whether there exists a group of applications for which a mixed language is the solution. Multi-disciplinary optimization problems contain data parallelism within a larger problem. For example, DNA protein sequence comparison [8] contains this type of parallelism. These examples are by no means exhaustive - merely representative. One difficulty with identifying applications is that the tools available influence how, and which, applications are coded. Recent activity in the field of combined task and data parallel language design [3, 5, 7, 22] is further evidence of the importance of the problem.

We have designed a set of data parallel extensions to the Mentat Programming Language (MPL) which allow us to integrate task parallelism, data parallelism, and nested task and data parallelism within a single language on top of a single run time system. The result is an object-oriented language, Braid, that supports both task and data parallelism on MIMD machines. The data parallel extensions include both element-centered operations as well as a more general notion of subset-centered operations which has not been

This work was partially funded by the General Electric Faculty for the Future Program and NSF grants ASC-9201822 and CDA-8922545-01.

To appear: Frontiers 95 - The Fifth Symposium on the Frontiers of Massively Parallel Computation, Feb. 6 - 9, 1995, McLean, Virginia.
Available as: University of Virginia Department of Computer Science Technical Report CS-94-45.

defined previously. These operations permit member functions to be defined and executed in parallel on structured subsets of the data. Based on programmer annotations, iteration constructs are automatically generated by the compiler. Data parallel operations are not limited to standard array or pre-defined operations. Rather the programmer is allowed to define complex data parallel operations for a data set.

In this paper we focus on the language design. Complete language details and a description of the translations from the source language to the target task parallel run-time system can be found in [27]. We begin with background material on data parallel computation and Mentat to set the context for this work. The extensions are then presented, followed by examples that illustrate the integration of task and data parallelism. We conclude with related work, a summary, and the status of the project.

2: Background

2.1: Data Parallelism

A data parallel computation is characterized by a particular data set whose elements have the same basic properties[2]. For example, a 1024x1024 image will have approximately a million elements, each of which has the same representation (say, an integer). Computations which manipulate this data set involve the simultaneous application of an operation to the elements of the data set.

In order to clarify the notion of a data parallel operation, we present some specific examples and order them by increasing complexity.

1. *Scalar Operations:* A scalar addition is performed on every element of the data set.
2. *Neighbor Operations:* A neighbor operation involves updating each element of the data set using its neighboring values, e.g., image convolution and iterative PDE solvers. Synchronization issues are important for these operations to produce deterministic results.
3. *Simple Array Operations:* A simple element to element predefined operation (addition) between two data sets. The complexity arises due to the use of data parallel objects that may not be correctly aligned with each other or properly distributed among processing resources.
4. *Complex Array Operations:* In matrix multiplication, each element of the result data set is the dot product of a row of one matrix and a column of another. Each of the dot products is independent of the others, and can be performed in parallel.
5. *Non-traditional Data Parallel Operations:* The data parallel style can also be applied in other domains such as gene sequence comparison. In this case, each element (gene sequence) is a string of characters. These elements form a data set (sequence library) and each element is compared against a single unknown element (gene sequence) using heuristic methods.

All of these examples are amenable to data parallel solutions because the same operations are performed on each element of the data set. Note that the operations range from simple scalar addition, to regular but computationally expensive dot products, to complex heuristics.

Specifying a data parallel computation in terms of a single element is the approach we have used in creating our data parallel extensions. We call this approach *element-centered*. Fundamentally, this concept is not new to data parallel languages [2, 4, 6, 13, 15, 16, 17, 20, 21]. However, we have extended the notion to encompass *subset level* data parallelism. By subset level data parallelism we mean allowing the definition of operations in which subsets (as opposed to elements) are the data granules, e.g., a row or a column. This is not the same as applying an element-centered operation to a subset of the elements as in enhancing a portion of an image. Subset-level data parallelism subsumes traditional element-level data parallelism as a special case and is essential in allowing for arbitrary, user-defined data parallel operations. We give an example in Section 3.3.

2.2: The Mentat Programming Language

Mentat is an object-oriented parallel processing system designed to provide: 1) easy-to-use parallelism, 2) high performance via parallel execution, and 3) applications portability across a wide range of platforms. Mentat has been ported to a variety of MIMD platforms and has been used to implement real-world applications in industry, government, and academia.

The Mentat Programming Language (MPL) [19] is a task parallel language based on C++ [23]. The Mentat approach exploits the object-oriented paradigm to provide high-level abstractions that mask the complex aspects of parallel programming, communication, synchronization, and scheduling from the programmer. The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution. Objects (tasks) may be created dynamically, and communication is implicit across method boundaries. The granule of computation is the Mentat class member function. A Mentat class is specified by using the keyword “mentat” as a prefix to class definitions. Member function invocation on Mentat objects is syntactically the

[2] This characterization has been true in the past, however, the emergence of nested data parallelism may cause a realignment of this view. Here we will assume that all elements of a data set are structurally equivalent.

```
 1:  dataparallel mentat class image{
 2:         //private member variables - “single element”.
 3:         int pixel;
 4:         float another_pixel;
 5:      public:
 6:         // public member functions - data parallel methods.
 7:         void AGG scale ELEMENT (int value);
 8:         void OVR overlay_pixel (RMAJ int *pixel_data);
 9:         int RED min_elem();
10:  };
11:
12:  void AGG image::scale ELEMENT (int value){
13:         pixel = pixel * value;
14:  }
15:
16:  int *value;
17:
18:  my_image.scale(10);              /* Invocation on a array */
19:  my_image[1][].scale(10);         /* Invocation on a row */
20:  my_image[][1].scale(5);          /* Invocation on a col */
21:  my_image[4][4].scale(10);        /* Invocation on an element */
```

Figure 1 Data parallel class and method definition. The keywords **AGG**, **OVR**, and **RED** specify the type of member function. **ELEMENT** specifies the subset type. Other subset types include **ROW** and **COL**.

same as for C++ objects. Exploiting opportunities for parallelism, communication, synchronization, and scheduling are all handled by Mentat.

The features of the MPL support a task parallel programming model in very natural way. Data parallel computations can be relatively easily mapped by hand onto the medium grain task parallel model supported by Mentat. However, the burden of managing the distribution of data rests with the programmer, as does the generation of iteration code to loop over contained elements, and other mind-numbing details. Our data parallel extensions are designed to relieve the programmer of these details. The resulting language supports both task and data parallelism.

3: Data Parallel Extensions to the Mentat Programming Language

3.1: Data Parallel Class Definition

The definition of a *data parallel mentat class* is the primary means by which the programmer specifies the domain of data parallel computation. Figure 1 illustrates a simple example of a data parallel class and conveys the idea of an element-level approach to data parallel class definition. Data parallel mentat classes are designated by pre-pending the keywords **`dataparallel mentat`** in front of the C++ keyword **`class`**. The structure of the data parallel class is similar to a C++ class in that member variables describe the data managed by the member functions of the class.

The member variables of the class definition represent a single element of the data set. This “logical” element is a template for the aggregate elements of the data set. In our example an element has two components, an integer value (`pixel`) and a floating point value (`another_pixel`). Any primitive or user defined type is allowable as a member variable of an element. Note that the programmer does not specify the size of the data set in the data parallel mentat class definition. When an instance of the class is instantiated the actual number of elements is specified. Once created, the elements can be referred to individually, in subsets, or as a whole.

The method definitions differ from those of C++ classes. The member functions of a data parallel class are annotated as shown in Figure 1 to convey information to the compiler. Three types of member functions may be defined for Braid data parallel classes: *overlay* functions, which are used to initialize the data parallel object, *aggregate* functions, which are applied to all elements (or subsets) of the data parallel object, and *reduction* functions, which allow the programmer to distill certain information from the values of the data set. The types are distinguished by the annotations **OVR**, **AGG** and **RED** respectively. All types of data parallel methods may have local variables and arguments of primitive or user defined types[3]. These variables are used in the same manner as C++ arguments and local variables. As with regular C++ classes, data parallel member functions are defined within the class definition as shown in Figure 1, lines 7-9. Figure 2 gives a brief grammar that defines the annotation syntax for each type. Because of

[3]. Currently, we do not allow data parallel objects as local variables for data parallel methods or as member variables of the elements. Doing so will be one step in supporting nested data parallelism.

<dp_mbrfcn>:	<return_type> **AGG** <fcn_name> <subset_specifier> ([<arg> | <agg_arg>], <arg>*); |
	void **OVR** <fcn_name> ([<arg> | <ovr_arg>]*); |
	<return_type> **RED** <fcn_name> (<arg>*);
<arg>:	any C++ expression
<agg_arg>:	<subset_specifier> <combination_rule> <dp_operand_type> <dp_operand_name>
<ovr_arg>:	<major_order_ind> <operand_type> <operand_name>
<subset_specifier>:	**ELEMENT** | **ROW** | **COLUMN**
<combination_rule>:	**1x1** | **1xN** | **Nx1**
<major_order_ind>:	**RMAJ** | **CMAJ**

Figure 2 Grammar for data parallel object methods.

space limitations we will confine our discussion here to the more interesting aggregate operation.

3.2: Simple Aggregate Methods

An aggregate function is applied to every element (or subset) of the data set. The simplest kind of data parallel operation, of which `scale()` (line 7 of Figure 1) is a good example, is one in which each individual element receives the same "treatment". However, there are often cases in which the programmer may wish to apply an operation to every row instead of every element, i.e. the operation only has meaning when applied to a row. In this case, the data must be split into subsets and each subset is then treated as a unit during the operation. This is the essence of subset level data parallelism and is achieved in Braid by defining data parallel methods which express operations on subsets as opposed to simply elements. (Obviously, elements are the base case.)

There are two steps required to define a data parallel operation within a data parallel class. First, the function must be annotated to indicate the type of the data parallel operation and the subset size to be used as the unit of operation. Second, the function must be written in a "location independant" manner.

The first production of the `<dp_mbrfcn>` non-terminal in Figure 2 shows the grammar for annotating this type of function. In this case, the `scale()` function is an aggregate function as designated by the annotation **AGG**. The `<subset_specifier>` annotation (shown in the grammar), in combination with the instructions of the actual function, allows the programmer to make this distinction. The annotation **ELEMENT** for `scale()` indicates that the method should be applied to each element of the data set and that our subset size is a single element. Similarly, a **ROW** annotation would imply application to each row of the data set.

In our approach, the body of the member function must be written as if the operation is being performed upon a single subset. In order to write subset-level operations, the programmer is responsible for providing any iteration required within the subset while the compiler provides iteration across the subsets. For example, a **ROW** specifier denotes that the programmer will specify the iteration within a row of the data parallel object, while the compiler will specify the iteration across rows of the data parallel object. For **ELEMENT** annotations iteration specification is not required of the programmer.

Current data parallel languages allow the programmer to invoke array operations without requiring the programmer to supply iteration control. However, for a complex array operation other than a language defined primitive, the burden is on the programmer to provide all iteration control within the data set. Our annotations allow the programmer to concentrate on higher level aspects of program development and shift much of the responsibility for iteration control to the compiler for primitive and complex data parallel operations.

Location independence is the second requirement for the definition of data parallel operations. This characteristic requires that the body of the function be written without reference to any specific element. In `scale()` the member variable `pixel` is assigned its value multiplied by the argument `value`. No reference is made by the programmer to a particular element such as `object[5][10].pixel`. The compiler will use the information contained in the annotations and the invocation of the method to index into the data set properly.

Mechanism is provided to allow the programmer to reference other elements of the data set in a relative fashion. The starting point is the element to which the operation is being applied, and the relative addresses are resolved at run-time. For example, `W()->pixel`, or `W()->S()->pixel`. The programmer may also specify boundary conditions as well. These mechanisms are similar to [21].

Invocation of an aggregate operation is shown on lines 18-21 of Figure 1. Assume that a data parallel object identified my_image has been declared as a two dimensional data parallel object of type image. The invocation syntax is exactly the same as the C++ invocation of a member function. The compiler ensures that the iteration within the operation is restricted to the elements indicated by the invocation. Therefore, the invocation on line 18 will result in an application of scale() to every element of my_image; each element of the second row of my_image will be operated upon as a result of the invocation on line 19; finally, the fifth element of the fifth row will be the only element of my_image to be scaled as a result of the invocation on line 21.

A primitive or user defined type may serve as the return type of a data parallel class member function. Currently, data parallel objects are not allowed as return values. Again, the notion of an element-level operation comes into play. Since the programmer is specifying the action of the operation in terms of one element (or subset), then the logical return result will be a single result per element (or subset). The actual number of results returned is specified completely by the annotations and the invocation which indicate the number of subsets in the invoked object. The compiler arranges for the allocation of the proper amount of space given the number of actual results to be returned. For example, suppose that our scale() method returned an integer. Then the invocation on line 18 would cause space for *k* integers to be allocated, where *k* = *n x m*, the dimensions of the invoked object. The variable to which the result of the function is to be assigned then points to this newly allocated space.

3.3: Complex Aggregate Methods

A data parallel object may also serve as a parameter to an aggregate operation. Data parallel arguments require two annotations which indicate the subset size and the manner in which subsets of the argument object and subsets of the invoked object will be combined. Note the syntax in Figure 2 and the dot product example in Figure 3. The first of these annotations is identical in form and meaning to the <subset_specifier> annotation described above for invoked objects. The annotation is associated with the data parallel object listed as the actual parameter, and indicates the portion of the argument object for which the programmer will provide iteration. Again, this portion must be treated in a "location independent" manner within the data parallel function body.

The second annotation, a <combination_rule>, indicates how the subsets of the operand will be combined with the subsets of the invoked object. By combined we mean the subsets are operands for a particular operation and are "combined" to form a result. For instance, the programmer may want to have each subset of the invoked object combined with a corresponding subset of the argument object. Matrix addition is an example of this type of combination, and is indicated with a **1x1** <combination_rule> annotation. Alternatively, the desired functionality may be to combine one subset from the invoked object with every subset of the argument object, or vice versa. These options may be indicated with the **1xN** and **Nx1** annotations respectively. An intuitive example of this type of combination is a dot product where every row of one matrix is com-

```
dataparallel mentat class matrix{
        //private member variables that specify an element.
    public:
        // Member functions for matrix multiplication and +=
        float AGG dot_product ROW (COL 1XN matrix& B);
        void AGG operator+= ELEMENT (ELEMENT 1X1 matrix&B);
}
float AGG matrix::dot_product ROW (COL 1XN matrix B) {
    float result;
    for (int j = 0; j < this.num_cols(); j++)
        result += this[j].value * B[j].value;
    return(result);
}
main()
{
        matrix *matrix_A, *matrix_B;
        matrix_A = new (8, 8, 4PT, 2, matrix_B, dot_product()) matrix ();
        matrix_A+=matrix_B;
        float *c = matrix_A.dot_product(B);
}
```

Figure 3 Data parallel object declaration, communication pattern specification, and complex aggregate method declaration and use. Allocation and initialization of matrix_B is not shown.

bined with every column of another matrix. Figure 3 shows the method definition for dot product and illustrates the use of `<subset_specifier>` and `<combination_rule>` annotations.

3.4: Object Creation and Distribution

Data parallel objects must be explicitly created using the operator `new()`. The programmer is required to specify the size and dimensions of the data parallel object, and optionally to specify both local and non-local communication patterns. This information is used by the run-time system to allocate data items to processors in such a way that communication between processors is reduced. Line 17 of Figure 3 demonstrates the creation of a data parallel object.

The arguments used to overload `new()` fall into three categories which we will refer to as *dimensions*, *local communication*, and *non-local communication*. Each category encompasses two of the arguments to `new()`. For the *dimension* category, the first two arguments specify the size of the data parallel object in the row and column dimensions respectively, an 8x8 array in the example. The argument values may be expressions.

For the *local communication* category the third and fourth arguments enumerate the type of communication that will be dominant within the data parallel object. Local communication occurs in terms of data parallel objects rather than processors, an example is a neighbor averaging function. Types of local communication that may be indicated by the programmer are **`NONE`**, **`PRED-SUCC`**, **`NS`**, **`EW`**, **`4PT`**, and **`8PT`**. **`PRED-SUCC`** applies to vectors while **`NS`**, **`EW`**, **`4PT`**, and **`8PT`** apply to two dimensional arrays. **`NONE`** applies to both vectors and two dimensional arrays. The local communication characteristic is conveyed in the first argument of the second set. The radius argument is the second argument in the set, and simply specifies the radius of the local communication (the value must be nonnegative). Examples of local communication patterns with a radius greater than one are shown in Figure 4.

The final category, *non-local communication*, provides information about interactions between the data parallel object being created and other data parallel objects of the application. The fifth argument to the `new()` operation indicates the class which is most often used as the actual argument to the method specified by the sixth argument. The sixth argument to the `new()` operation specifies the dominant function of the data parallel object being created. This information serves to identify the dominant method of the object, and thus where the majority of time is spent during execution. The distribution and alignments of the objects are made with this information in mind. Once the proper amount of space is determined, the compiler establishes a distribution pattern for the elements across the processors. This is accomplished in the body of the overloaded `new()` operation.

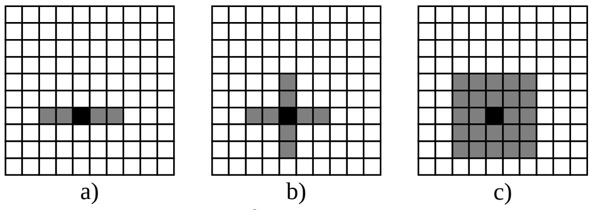

Figure 4 Three types of local communication pattern each with a radius of two.The darkly shaded element is the element to which the neighboring values are “communicated”. a) EW pattern. b) 4PT pattern. c) 8PT pattern.

A number of methods may be defined for the data parallel class, and all methods may not exhibit the same local or non-local communication pattern. Thus, in order to achieve the best performance, the programmer should indicate the pattern and methods that will be used most often by the object.

The compiler passes the information concerning the number and distribution of elements to be allocated for the object being created to the run-time system. These hints indicate the general form of the decomposition, alignment, and distribution of the data among processors. Joint research is now being conducted within the Mentat group to allow the run-time system to combine the compiler generated information with information about the current machine architecture. The run-time system will employ heuristic algorithms to automatically handle the decomposition, distribution and alignment of the data parallel object [25, 26]. Cooperating with the run-time system in this manner allows us to divorce ourselves from the underlying machine architecture. This flexibility is critical in order to exploit a heterogeneous system architecture.

We believe that decomposition, distribution and alignment as described in Fortran-D and other data parallel languages are equivalent to our mechanisms of specifying the communication patterns. These languages work best when the programmer to knows the topology of the underlying processors and interconnection network in advance. In contrast, our extensions allow the programmer to simply indicate which method will dominate the computation. This serves to remove not only the job of data placement from the programmer, but to also alleviate the need for the programmer to explicitly reference off-host data.

3.5: Deterministic Data Parallel Semantics

The main characteristic of data parallel computation is that the operation being applied to the data set is applied “logically simultaneously” to every element of the data set. If there are enough physical processors to manage one element per processor, then the operation will truly proceed in parallel across the data set. In most cases though, each processor will be responsible for managing a number of data

```
1:  ...
2:  float x, z;
3:  int y;
4:
5:  control_parallel_obj A, B,C, D, E;
6:  // B.op1() internally uses dataparallel classes
7:
8:  x = A.op2(4);
9:  z = B.op1(x);
10: y = C.op2(x);
11: y = D.op2(y);
12: y = E.op3(y,z);
```

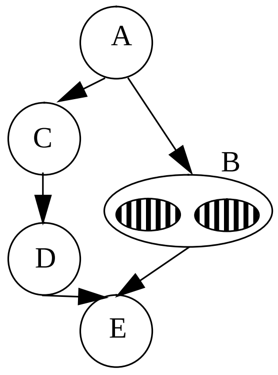

Figure 5 A task parallel Mentat code fragment. The resulting macro-dataflow graph is shown on the right. The implementation of B.op1() "manages" two dataparallel objects. The circles represent task parallel method invocations, the hatched ovals represent data parallel computations, and the arcs represent data dependencies. The graph is constructed at run-time using compiler generated information.

elements. When multiple data are assigned to a single processor, the degree of parallelism decreases, and the potential for race conditions on member variables is introduced. Consider for example, a neighbor averaging function. The execution order of operations to the data items could affect the outcome. We desire a deterministic semantics for our extensions. Thus, the computation should proceed *as if* there were as many processors as elements. To achieve this, each processor must iterate through the data elements for which it is responsible, applying the operation to each in turn. To address these synchronization issues within a data parallel object we have chosen a pre-copy deterministic semantics. For these semantics, local updates of an element are not visible to any other element in the data set until the operation is complete.

With respect to operations invoked upon a data parallel object, we base the implementation of data parallel objects on the *sequential persistent* mentat class. This type of class orders all invocations by a caller upon the invoked object. Note that the MPL and the Mentat run-time system also supports *persistent* mentat classes with monitor-like properties which are especially useful for control parallelism.

4: Integration of Task and Data Parallelism

The integration of task and data parallelism is the primary motivation behind Braid. At the program level this integration is seamless, task and data parallel objects are invoked in the same manner as regular C++ objects. The encapsulation properties of the object-oriented paradigm allow the programmer to manipulate objects at the program level without being concerned about the details of how parallelism is achieved within the object. Thus, the programmer may focus on higher level program design issues once the data and task parallel classes have been constructed.

Parallelism encapsulation allows for two types of integrated task and data parallelism. First, concurrent execution of task and data parallel objects, and second, management of concurrent data parallel objects by a task parallel "manager".[4] From an implementation perspective, the Mentat Run-Time System monitors the use of results of mentat object member function invocations of task and data parallel objects and constructs medium grain program graphs [11, 12].

We demonstrate the effect in Figure 5 and Figure 6. The first example illustrates the use of encapsulated data parallelism in which a task parallel object member function invokes a data parallel object in the course of execution. This "encapsulated" data parallelism is completely hidden from the invoker. The results of the object member functions are x, y, and z. The semantics allow the method calls on lines 8-12 to be executed in parallel. On line 12, the result of the data parallel method is used as an argument to a task parallel method. Furthermore, the entire code block itself could be a mentat object member function implementation executing concurrently with other mentat object member functions. Figure 6 illustrates both task and data parallel invocations within the same code fragment.

5: Related Work

In terms of purely data parallel languages, Dataparallel C [13, 20, 21], pC++ [2, 16], C** [15], Fortran D [6], Fortran 90 [1], and High Performance Fortran (HPF) [17] are the languages from which are related to our work. C** and pC++ are based on C++. Dataparallel C is based on C, but uses some ideas from object-oriented language design. HPF's origin is Fortran. Our work differs from previous work in methods for specifying data distributions, supporting subset data parallelism, and allowing arbitrary, user-

[4] We believe that using task parallel objects as the "elements" of a data parallel class is feasible as well, but must explore various implementation issues first.

```
dataparallel mentat class data_parallel_obj {
       // private member variables
   public:
       // public member functions
       int AGG row_sums ROW ();
       ...
}

...
float x, z;
int y;

control_parallel_obj A, B;
data_parallel_obj my_image;

x = A.op1();
y = my_image.row_sums();
z = B.op1(x,y);
```

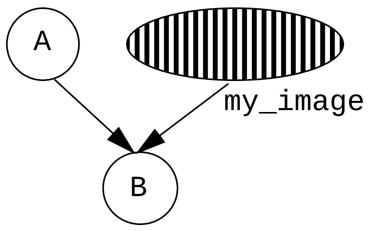

Figure 6 Control and data parallel method invocations in the same code block. The circles represent task parallel method invocations, the hatched ovals represent data parallel computations, and the arcs represent data dependencies.

defined data parallel operations.

Currently, there are two groups which have reported work on integrated task and data parallelism. Both are Fortran based. Subhlok et al. [24] describe a parallelizing Fortran 77 based compiler augmented with Fortran 90, Fortran D and HPF constructs whose target is the iWarp system. The motivation for the work is exploring the trade-off between pure data parallelism and groups of pipelined data parallel operations organized into tasks. Our language differs in that task parallel loops are not restricted to constant bounds and communication is implicit. Further, our underlying model, MDF [11], supports both dynamic program graphs and persistence rather than a static pure data mode. In terms of data parallelism, our work differs as mentioned above with respect to the various Fortran dialects. Finally, Subhlok et al. are adding task parallel constructs to existing data parallel language mechanisms. We begin with a task parallel paradigm and incorporate data parallelism. Braid also allows execution of task and data parallel components at the same level, while Subhlok presents data parallelism contained within task parallelism.

Foster et al. [5] and Chandy et al. [3] describe an interface for combining Fortran M and HPF to achieve the integration of task and data parallelism. Their approach differs from our own and Subhlok's in that the mechanism used is actually two separate compilers, one task parallel and one data parallel. Eventually they expect to produce a single language and compiler. Again, our work differs in that for task parallelism their communication is explicit while ours is implicit and we allow the definition of arbitrary, user-defined data parallel operations. Also, they assume programmer knowledge of machine architecture and the number of processors available to the application. Both Subhlok and Foster and Chandy have produced preliminary performance data.

The overriding difference between our work and that mentioned above is our basis in C++ and the object-oriented approach. We are aware of efforts by Gannon [7] and Quinn [22]. The base languages of these projects are C++ and C respectively, however, no results combining task and data parallelism have been reported at the time of this writing.

6: Status and Future Work

Braid integrates both task and data parallelism into a single language design. The language is intended to satisfy the needs of applications requiring a mix of the two paradigms. The integration was accomplished by extending an existing task parallel language, the Mentat Programming Language (MPL), to include data parallel mentat classes. The use of data parallel mentat classes can be combined with standard mentat classes to produce a mixed task parallel and data parallel application. Additionally, we have introduced the idea of subset parallelism which allows the user to define complex data parallel operations across subsets of a data set.

In [27] the language is described in detail, as are the translations of the data parallel extensions to the underlying task parallel run-time system. The current version of the MPL compiler is operational and has been used to develop a number of real-world applications [8, 10]. While hand translations have been performed for Braid's language features, the data parallel portion of the compiler is not yet complete. We are working on the required compiler support and on investigating the use of nested data parallelism. In our current language model, the dominant data distribution is specified upon creation of an object. We intend to allow the programmer to redefine these dominant distributions at run-time in future versions of the language. We anticipate that an efficient implementation will be the most difficult hurdle in providing this type of dynamic redistribution. We also intend to explore the issues involved in moving beyond traditional array data parallelism to supporting data set organizations such as trees and unstructured groups. Finally, we expect to evaluate the performance of mixed parallelism applications developed using our approach.

7: References

[1] American National Standards Institute 1990. ANSI X3J3/S8.115. Fortran 90.

[2] F. Bodin et al., "Distributed pC++: Basic Ideas for an Object Parallel Language," *Proceedings Object-Oriented Numerics Conference*, April 25-27, 1993, Sunriver, Oregon, pp. 1-24.

[3] K.M. Chandy et al., "Integrated Support for Task and Data Parallelism", *Intl. J. of Supercomputer Applications*, 8(2), 1994, pp. 80-98.

[4] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, Vol. 1, No. 1, Aug. 1992, pp. 31-50.

[5] I. Foster, M. Xu, B. Avalani, A. Choudhary, "A Compilation System that Integrates High Performance Fortran and Fortran M", Proccedings of the 1994 Scalable High Performance Computing Conference.

[6] G.C. Fox et al., "Fortran D Language Specifications," Technical Report SCCS 42c, NPAC, Syracuse University, Syracuse, NY.

[7] D. Gannon, *personal communications*, 1993, 1994

[8] A.S. Grimshaw, E.A. West, W. Pearson, "No Pain and Gain! - Experiences with Mentat on a Biological Application," *Concurrency: Practice and Experience*, 5(4), June 1993, pp. 309-328.

[9] A.S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, May, 1993, pp. 39-51.

[10] A.S. Grimshaw, W.T. Strayer, and P. Narayan, "Dynamic, Object-Oriented Parallel Processing," IEEE Parallel and Distributed Technology, 1(2), May 1993, pp. 33-47.

[11] A.S. Grimshaw, "The Mentat Computation Model - Data-Driven Support for Dynamic Object-Oriented Parallel Processing," Technical Report CS-93-30, University of Virginia, Computer Science Department, Charlottesville, VA, 1993.

[12] A.S. Grimshaw, J.B. Weissman, and W.T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing," to appear *ACM Transactions on Computer Systems*.

[13] P.J. Hatcher et al., "Compiling Data-Parallel Programs for MIMD Architectures," *European Workshop on Parallel Computing*, March 1992, Barcelona, Spain.

[14] J.F. Karpovich et al., "A Parallel Object-Oriented Framework for Stencil Algorithms," *Proceedings of the Second Symposium on High-Performance Distributed Computing*, July, 1993, Spokane, WA, pp. 34-41.

[15] J.R. Larus, B. Richards, and G. Viswanathan, "C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language," Technical Report 1126, University of Wisconsin, Computer Science Department, Madison, Wisconsin, 1992.

[16] J.K. Lee and D. Gannon, "Object Oriented Parallel Programming Experiments and Results," *Proceedings of Supercomputing '91*, 1991, Albuquerque, NM, pp. 273-282.

[17] D.B. Loveman, "High Performance Fortran," *IEEE Parallel & Distributed Technology: Systems & Applications*, Vol. 1, No. 1, Feb., 1993, pp. 25-42.

[18] C.R. Mechoso, J.D. Farrara, J.A. Spahr, "Running a Climate Model in a Heterogeneous, Distributed Computer Environment," *Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing*, April 2-5, 1994, San Francisco, California, pp. 79-84.

[19] Mentat Research Group, "Mentat 2.5 Programming Language Reference Manual," Technical Report CS-94-05, University of Virginia, Department of Computer Science, Charlottesville, VA, 1994.

[20] N. Nedeljkovic and M.J. Quinn, "Data-Parallel Programming on a Network of Heterogeneous Workstations," *Concurrency: Practice and Experience*, 5(4), June 1993, pp.257-268.

[21] M.J. Quinn and P.J. Hatcher, "Data-Parallel Programming on Multicomputers," *IEEE Software*, Sept. 1990, pp. 69-76.

[22] M. J. Quinn, *personal communication*, 1992.

[23] B. Stroustrup, *The C++ Programming Language*, 2nd ed. Addison-Wesley, Reading, Mass., 1991.

[24] J. Subhlok, J.M. Stichnoth, D.R. O'Hallaron, and T. Gross, "Exploiting Task and Data Parallelism on a Multicomputer," *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May, 1993, San Diego, CA, pp. 13-22.

[25] J.B. Weissman and A.S. Grimshaw, "Multigranular Scheduling of Data Parallel Programs," Technical Report CS-93-38, University of Virginia, Department of Computer Science, Charlottesville, VA, July, 1993.

[26] J.B. Weissman and A.S. Grimshaw, "Network Partitioning of Data Parallel Computations," *Proceedings of the Symposium on High-Performance Distributed Computing (HPDC-3)*, August, 1994, San Francisco, CA, pp. 149-156.

[27] E. A. West, "Combining Control and Data Parallelism: Data Parallel Extensions to the Mentat Programming Language,", Technical Report CS-94-16, University of Virginia, Department of Computer Science, Charlottesville, VA, May, 1994.

Information regarding current work on Braid can be found at http://uvacs.cs.virginia.edu/~eaw2t/Braid.html. For information on Mentat and the Mentat Programming Language see http://uvacs.cs.virginia.edu/~mentat/.