Combining Control and Data Parallelism: Data Parallel Extensions to the Mentat Programming Language

Emily A. West

Computer Science Report No. CS-94-16
May 18, 1994

Combining Control and Data Parallelism: Data Parallel Extensions to the Mentat Programming Language

Emily A. West
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
west@virginia.edu

Abstract

Control parallelism refers to concurrent execution of different instruction streams. Data parallelism refers to concurrent execution of the same instruction stream on multiple data. There are a number of languages which support control parallelism as well as several which support data parallelism. As yet, there is no language which combines the two. The Mentat Programming Language, MPL, is designed to express control parallelism. While expression of data parallelism is possible, it is awkward and unsupported by the current language features. In this research, we propose a set of data parallel extensions to the MPL. We define a new type of mentat class, the *dataparallel mentat class*, to complement the existing *regular, persistent* and *sequential* mentat classes. In a dataparallel mentat class, the programmer defines the structure of an element and the methods that operate on these elements. These methods are annotated to convey the distribution of the data set and inter and intra data set communication. All other data parallel languages to date simply allow element level parallelism. Our language extensions support subset level parallelism as well. In this work, we present the language design, a description of the implementation model, and the translations of the dataparallel class to Mentat's control parallel model.

Combining Control and Data Parallelism: Data Parallel Extensions to the Mentat Programming Language

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science (Computer Science)

by

Emily Archer West

May 1994

APPROVAL SHEET

This thesis is submitted in partial fulfillment of the
requirements for the degree of
Master of Science (Computer Science)

Emily A. West

This thesis has been read and approved by the Examining Committee :

Thesis Advisor: Andrew S. Grimshaw

Committee Chair: William A. Wulf

Committee Member: Jack W. Davidson

Accepted for the School of Engineering and Applied Science:

Dean Edgar A. Starke, Jr.
School of Engineering and Applied Science

May 1994

Abstract

Control parallelism refers to concurrent execution of different instruction streams. Data parallelism refers to concurrent execution of the same instruction stream on multiple data. There are a number of languages which support control parallelism as well as several which support data parallelism. As yet, there is no language which combines the two. The Mentat Programming Language, MPL, is designed to express control parallelism. While expression of data parallelism is possible, it is awkward and unsupported by the current language features. In this research, we propose a set of data parallel extensions to the MPL. We define a new type of mentat class, the *dataparallel mentat class*, to complement the existing *regular*, *persistent* and *sequential* mentat classes. In a dataparallel mentat class, the programmer defines the structure of an element and the methods that operate on these elements. These methods are annotated to convey the distribution of the data set and inter and intra data set communication. All other data parallel languages to date simply allow element level parallelism. Our language extensions support subset level parallelism as well. In this work, we present the language design, a description of the implementation model, and the translations of the dataparallel class to Mentat's control parallel model.

Dedication

For my husband, Charlie.

Acknowledgments

I thank my advisor, Andrew Grimshaw, for his continuing faith in the value of this work and for his unflagging support throughout. General Electric’s financial support through the Faculty for the Future program made it possible for me to concentrate on this work for the past year. The love and support of my parents has been extremely important to me throughout my life. I will be forever indebted to Ezmerelda for providing me with the hardest deadline anyone could ever imagine. Finally, my most sincere gratitude is reserved for my husband Charlie, whose love and encouragement are without bound.

Table of Contents

List of Figures

Chapter 1 Introduction

Control parallelism refers to concurrent execution of different instruction streams. Data parallelism refers to concurrent execution of the same instruction stream on multiple data. Significant applications exist that contain both control and data parallel components. Global climate and geophysical models are examples of multi-level models of natural phenomena which exhibit mixed forms of parallelism. Likewise, the progression from image processing to feature extraction and finally image understanding presents many opportunities for the exploitation of both control and data parallelism.

Current parallel languages support either control parallelism or data parallelism, unfortunately we are not aware of any languages which support both. Given this situation, developers of applications exhibiting mixed parallelism are faced with one of two choices. The first alternative is to write the control parallel components using an existing control parallel language, and develop the data parallel components using a data parallel language. These components will most likely be distinct executables. The downside of this solution is that the programmer must provide an interface between the resulting components. This is a tiresome task at best. The second option for this type of application development is use one language type to exploit the corresponding type of parallelism within the application. This solution is to the detriment of the remaining application components which cannot be easily expressed using the chosen language. Given these options, the need for programming environments (a language, compiler and run-time system) to support development of applications containing mixed parallelism is clear.

We address this problem at the language design level. Our objective is to provide a language in which both control and data parallelism are easily expressible and readily used in conjunction with one another. We begin with a control parallel language, the Mentat Programming Language (MPL), and extend its features to support the data parallel style of programming. The MPL is an object-oriented parallel programming language which provides control parallelism at the method level. Data parallel applications have been implemented using this language[9]. However, many of the techniques used to develop these applications are automatically provided in current data parallel languages such as

Dataparallel C[11] and pC++[2]. A great deal of handcoding was required to achieve the same result with the MPL. We believe this lack of support is a deficiency of the language.

There are a number of data parallel languages in existence today. Our extensions build on the previous research done to develop these languages, however, we also generalize a number of known mechanisms, provide more flexibility in terms of data set manipulation, and incorporate several new features. In this thesis we describe the new language constructs which have been added to the MPL, an implementation model for these constructs and a description of the translations from our constructs to the implementation model.

As yet, we have not implemented a compiler for our language extensions, however, we believe efficient implementations exist for the code translations. This belief is justified by the work of Karpovich and Judd[13], who have implemented libraries which provide much of the functionality needed for our implementation, and by the pioneering work of Hatcher and Quinn [11]. In addition, the author conducted proof-of-concept experiments prior to the bulk of this research to test the feasibility of pursuing the chosen solution. For simplicity, our language only supports one and two dimensional array data structures. We accepted this limitation in order to focus on the larger issues of the language design. Incorporating more elaborate data set structures will be the target of future research.

The thesis is organized as follows. Chapter 2 includes a background discussion of data parallelism, the Mentat Programming Language, and presents a short description of a number of existing data parallel languages. The data parallel language extensions we have added to the Mentat Programming Language are related in Chapter 3. Chapter 4 discusses the implementation model for our data parallel extensions, and explains the translations needed to map the extensions to the control parallel paradigm of the Mentat system. We conclude in Chapter 5 with a discussion of our contributions and possible research avenues to be explored in the future.

Chapter 2 Background and Related Work

2.1 Data Parallelism

A data parallel computation is characterized by a particular data set whose elements have the same basic properties. For example, a 1024x1024 image will have approximately a million elements, each of which has the same representation (say, an integer). The elements are only distinguished by their particular values and/or relative ordering within the data set. Computations which manipulate this data set involve the simultaneous application of an operation to the elements of the data set. The multiplicity of data is responsible for the parallelism that can be extracted from the computation. Currently, there is little consensus in the literature concerning the specifics of data parallel operation semantics. This disparity is a result of the evolution of the data parallel style from an instruction level paradigm used to program SIMD (single-instruction, multiple-data) architectures into a more general style which encompasses MIMD (multiple-instruction, multiple-data) architectures as well. We delay detailed discussion of these semantic issues to Section 2.2 (Related Work).

In the original SIMD paradigm, parallel execution proceeds in lockstep at the instruction level for each element of the data set. This means that updated values of the elements are visible at instruction boundaries even when such synchronization is unnecessary. In the MIMD data parallel style, the SIMD style of lockstep instruction level execution produces too fine a granularity to translate to reasonable performance on a MIMD architecture. The style that has evolved is to form a coarser grained computation by distributing subsets of elements and have each processor iterate over its subset. However, in a distributed memory environment, logically simultaneous application of instructions must be enforced in some fashion.

Data parallel implementations on a MIMD architecture are typically written in an a different style as opposed to the strictly synchronous SIMD style. Generally, communication points in the program must be indicated by the programmer using send/ receive constructs. Therefore, synchronization occurs only when processes need to exchange data as opposed to after every instruction that is executed in parallel. The resulting style of programming is often called SPMD (single-program, multiple-data).

Because of its rigid structure and usually deterministic semantics, SPMD programming is often considered a conceptually easier than general control parallel programming. However, there are a number of tedious issues that the programmer must address. In addition to concurrent access and synchronization, the paradigm introduces issues such as data decomposition, distribution and alignment, data structure representation and addressing, and specialized data communication patterns. Data decomposition involves specifying the amount of data (in terms of individual data elements) that will be located on distinct processors. Data distribution deals with assigning data, or subsets of data, to a particular processor. Alignment is primarily (although not entirely) a Fortran artifact. In Fortran, structures are not allowed, and therefore multiple arrays must be used. In more modern languages a single array of a single user defined structure alleviates the need to align multiple arrays. The representation and organization of the data are crucial to generating an efficient data parallel computation. For this reason, a great deal of effort is often expended in designing efficient data structures, in particular minimizing the number of non-local accesses. Until recently, these tedious tasks were the programmer's responsibility. Recent attempts at a data parallel language design have attempted to automate these tasks for the programmer (see Section 2.3).

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

1. *Scalar Addition:* A scalar addition is performed on every element of the data set.
2. *Neighbor Operation:* A neighbor operation involves updating each element of the data set using its neighboring values. Examples include image convolution and the solution of PDE's using iterative methods such as Jacobbi iteration.
3. *Matrix Addition:* Matrix addition is done by a simple element to element addition between two data sets. The complexity arises due to the use of multiple data parallel objects that may not be correctly aligned.
4. *Matrix Multiply:* In matrix multiplication, each element of the result data set is the dot product of a row of one data set 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 previous examples are numeric matrix examples. 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.

With the exception of matrix multiply each of the above examples involves the application of a function to each element of the data set. Matrix multiply (**C**=**A******B**) is conceptually different. Rather than apply an operation in parallel to all elements of the set, the two input matrices can be thought of as being partitioned into subsets of rows and columns. Matrix multiply is the application of a dot-product operator applied to the structured subsets of the data parallel matrices. For each row in the A matrix apply dot-product to each column of the B matrix. Alternatively, for each subset of A, apply an operator to each subset of B. We call such data parallel formulations *subset data parallelism*. In matrix multiply the subsets are rows and columns. In general the subsets may be the individual elements as well. In that case subset parallelism subsumes traditional element level data parallelism.

The remainder of this chapter is divided into two sections. Section 2.2 is a brief overview of the Mentat programming language. Readers familiar with Mentat may wish to skip to the description of related work in Section 2.3.

2.2 The Mentat Programming Language

The three primary design objectives of Mentat are to provide: 1) easy-to-use parallelism, 2) high performance via parallel execution, and 3) applications portability across a wide range of platforms. The underlying premise is that writing programs for parallel machines does not have to be hard. It is the lack of appropriate abstractions that has kept parallel architectures difficult to program, and hence inaccessible to mainstream, production system programmers.

The Mentat approach exploits the object-oriented paradigm to provide high-level abstractions that mask the complex aspects of parallel programming, communication,

```
1:   mentat class bar {
2:   // private member functions and variables
3:   public:
4:       int op1(int,int);
5:       int op2(int, int);
6:   };
```

Figure 2.1 A Mentat class definition. Without the keyword “mentat”, it is a legitimate C++ class definition.

synchronization, and scheduling from the programmer. Instead of worrying about and managing these details, the programmer is free to concentrate on the details of the application. The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution. The complex tasks are handled by Mentat.

There are two primary components of Mentat: the Mentat Programming Language (MPL) [17] and the Mentat run-time system (RTS). The MPL is an object-oriented programming language based on C++ [20] that masks the complexity of the parallel environment from the programmer. The granule of computation is the Mentat class member function. Mentat classes consist of contained objects (local and member variables), their procedures, and a thread of control.

The most important MPL extension to C++ is the keyword “mentat” as a prefix to class definitions, as shown on line 1 of Figure 2.1. This keyword indicates to the compiler that the member functions of the class are computationally expensive enough to be worth doing in parallel. Mentat classes are defined to be either *regular*, *persistent*, or *sequential*. Regular Mentat classes are stateless, and their member functions can be thought of as pure functions in the sense that they maintain no state information between invocations. As a consequence, the run-time system may instantiate a new instance of a regular Mentat class to service each invocation of a member function from that class, even while other instances of the same function already exist.

On the other hand, persistent and sequential Mentat classes maintain state information between member function invocations. Since state must be maintained, each member function invocation on a persistent Mentat object is served by the same instance of the object. The difference between persistent and sequential classes is that invocations from a particular caller to a sequential mentat object are guaranteed to be executed in the order they were invoked. Invocations on persistent objects, on the other hand, will be executed as soon as the required data dependencies have been met. The result may be that the functions are executed in an order other than the invocation order.

Instances of Mentat classes are called *Mentat objects*. Each Mentat object possesses a unique name, an address space, and a single thread of control. Because Mentat objects are address space-disjoint, all communication is via member function invocation. Because Mentat objects have a single thread of control, they have monitor-like properties. In particular, only one member function may be executing at a time on a particular persistent object. Thus, there are no races on contained variables.

Variables whose classes are Mentat classes are analogous to variables that are pointers. They are not an instance of the class; rather, they name or point to an instance. We call these variables *Mentat variables*. As with pointers, Mentat variables are initially *unbound* (they do not name an instance) and must be explicitly *bound*. A bound Mentat variable names a specific Mentat object. Unlike pointers, when an unbound Mentat variable is used and a member function is invoked, it is not an error. Instead, if the class is a regular Mentat class, the underlying system instantiates a new Mentat object to service the member function invocation. The Mentat variable is not bound to the created instance.

2.2.1 Mentat Member Function Invocation

Member function invocation on Mentat objects is syntactically the same as for C++ objects. Semantically, however, there are three important differences. First, Mentat member function invocations are non-blocking, providing parallel execution of member functions when data dependencies permit. Second, each invocation of a regular Mentat object member function causes the instantiation of a new object to service the request. This, combined with non-blocking invocation, means that many instances of a regular class member function can be executing concurrently. Finally, Mentat member functions are always call-by-value because the model assumes distributed memory. All parameters are physically copied to the destination object. Similarly, return values are by-value. Pointers and references may be used as formal parameters and as results, but the effect is that the memory object to which the pointer points is copied. Variable size arguments are supported as well, since they facilitate the writing of library classes such as matrix algebra classes.

The features of the MPL support a task parallel programming model in very natural way. One can implement data parallel programs using task parallel Mentat objects. Each Mentat object contains, or manages, a set of smaller element objects. 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.

Details of the computation model (macro-data flow), the Mentat programming language and the Mentat run-time system can be found elsewhere [7, 8, 17].

2.3 Related Work

Dataparallel C [10, 11, 18, 19], pC++ [2, 15] C** [14] Fortran D [5] Fortran 90 [10] and High Performance Fortran (HPF) [16] are the languages from which we have borrowed ideas and which are related to our work. We have also developed some new ideas. C** and pC++ are based on C++. Dataparallel C is based on C, but uses some ideas from object-oriented language design. HPF's origins are obvious. All of these languages are strictly data parallel languages, and we differ from them all in that we are combining both control and data parallelism within the same language.

Dataparallel C

In [19], Quinn and Hatcher, the designers of Dataparallel C, show that a strictly SIMD program can be compiled to a distributed memory architecture without sacrificing performance. This entails loosening the synchronization points by synchronizing only when communication between physical processors is necessary. Data locality is also important, and they provide mechanisms for the programmer to convey this information to the compiler.

Dataparallel C is an extension of Thinking Machines C* language. The conceptual model presented to the programmer is a front-end and a back-end. Statements within the `domain select { stmt list; }` construct are applied to the data set in parallel by the back-end module. Synchronization between elements is only important within the domain statements, and is handled at the statement level. The compiler determines when synchronization is necessary instead of synchronizing after each instruction. The data set can be distributed among the processors, however, alignment and distribution are left up to the programmer. Each data set must be statically declared in a global name space.

Rather than the statement level synchronization semantics of Dataparallel C we support a deterministic pre-copy semantics with synchronization at member function boundaries. An advantage of our approach is that it is both deterministic and does not require heroic compiler efforts to minimize communication and synchronization. In addition we differ from Dataparallel C in that we allow not only element level parallelism within the data set, but subset parallelism. We provide alignment of multiple data sets, dynamic creation of data sets, and allow for the specification of general reduction methods as opposed to predefined reduction methods.

pC++

Lee and Gannon [15] have developed a distributed collection model and a C++ based language to support the model. The language provides mechanisms allowing the programmer to exploit memory locality, abstractions for implementing synchronous operations on distributed data structures, and mechanisms to express massive parallelism.

The model is based on Concurrent Aggregates[4] and allows them to de-couple the structure of the data from the element type. A collection, essentially a data set, is defined with the C++ class mechanism. Element types are defined in separate classes. Operations are applied concurrently to all elements of the collection. Invocation of parallel methods involves sending a message to a collection which then oversees the method's concurrent application to every element of the collection. The single control thread splits into multiple threads, the programmer is responsible for synchronization between threads and for resynchronization of the threads upon completion of the operation. A barrier call is provided for this purpose.

Distribution of data to take advantage of memory locality is done using a two step method in which collections are assigned the same abstract “template distribution”. The template distributions are then mapped to processors, effectively collocating the collections at the same processors.

There are several features of pC++ that might make the language difficult for the novice to master. First, the semantics are non-deterministic, the programmer must explicitly control potential race conditions. Races may occur for several different reasons; for example one processor may use an out of date value that another processor is updating, e.g., at a boundary region between processors. In fact programmers must be aware of whether a neighbor element is local or remote, and be aware of the different access semantics. A second difficulty is with the *MethodOfElement* functions. Each physical processor must iterate over the entire element space, to determine whether or not the element is local. Third, there are two views of a collection, as the whole object, explicitly indexed, or as elements. One cannot define natural decompositions such as rows and columns. The result is that the programmer is responsible for explicitly managing indexing, knowing the data's location, and enforcing dependencies (or living with the non-determinism caused by races).

In our work, we do not require the programmer to manage synchronization at any level. We allow the programmer more flexibility in the application of parallelism; we allow subset level parallelism as well as element level parallelism. We also provide distribution mechanisms for data sets.

*C***

Larus, the designer of C++ based C**, believes that previous data parallel languages reflect the lockstep synchronization of SIMD architectures too closely. He notes that these languages allow non-determinism with respect to updates of variables external to the scope of the elements. He claims that one advantage of data parallelism is the ability to reason about program behavior. Larus maintains that such reasoning is not dependant upon lockstep execution.

In C** all classes are parallel classes, and elements of the data set are limited to a single member variable. C**'s semantics preserve some of the properties of the SIMD programming model. Referenced data is copied to local scope before the execution of a parallel method. Upon completion of the method, all modified values are written back to their original location. Intermediate values are not visible to neighboring element computations. Up to this point, Larus' semantic techniques are the same as our own. However, C** allows multiple updates of the same global variable. The value that persists after the completion of an operation is non-deterministic. Mechanism is provided which allows the definition of subsets, however, these subsets must have their own explicitly defined methods. Methods of the data set may not be applied to any subset.

Fortran D

Fortran D is a data parallel extension of Fortran developed by Kennedy, Fox, and others [5]. It has many similarities to Vienna Fortran [3], which was simultaneously and independently developed. The basic idea in Fortran D is that data movement is costly and should be avoided. The programmer defines data structures (arrays) that are to be distributed to the processors. The programmer specifies the distribution of data to processors and specifies which data structures are to be aligned. The compiler then detects loops which are independent (DOALL loops) and generates code using the owner computes rule to move data to satisfy data dependencies. Fortran D has had a major influence on HPF Fortran. Many of the features of Fortran D have been directly incorporated into HPF.

Our work differs from Fortran D in several ways. Most importantly, our language supports both control and data parallelism by allowing one or more control parallel components to execute in parallel with a dataparallel operation. Second, alignment and distribution are de-emphasized. Instead we seek information on logical communication patterns from the user. Third, iteration is managed by the compiler, rather than being done by hand.

Fortran 90

While Fortran 90 is not a data parallel language, it is one of the first languages to support the notion of parallel array operations. For example, C = A + B, where A, B, and C are all equal dimension matrices of the same primitive type for which addition is defined. The programmer may not define operations between matrices, rather they are provided only for arithmetic operations on primitive data types.

High Performance Fortran

High Performance Fortran, HPF, builds on Fortran D and Fortran 90. It is a collaborative effort between industry, academia, and government. Several vendors have promised HPF compilers in the near future. HPF was specifically designed to be compiled onto massively parallel architectures. In addition to the array operations developed in Fortran 90, HPF provides mechanisms for the distribution of data across processors to take advantage of data locality in distributed memory architectures. They originally developed the two stage distribution mapping that is used by pC++.

Chapter 3 Data Parallel Extensions to the Mentat Programming Language

There are two characteristics of developing complex iterative control structures that can make data parallel programming a tedious endeavor. First, repetitious iteration over a (possibly large) data set is one of many tasks required of the data parallel application programmer. Often times, the for-loop structure of two separate data parallel operations is exactly the same. The only variations between the operations are the instructions executed at the innermost level. Thus, a great deal of work must be done that is not intellectually challenging. Secondly, at a glance, every for-loop is similar to every other for-loop. However, there may be a wide variation in the details of the for-loop structure of the operations, thus requiring the programmer to pay close attention to each new operation written. Any experienced C programmer who has been bitten by the <= vs. < bug can attest to this fact. Therefore, the time spent making sure a data parallel operation is correct is often not proportional to the actual complexity of the function being developed.

One of the primary methods of increasing the performance of a data parallel application is to distribute the data among a set of processors in such a way as to minimize communication. Managing this task by hand can rapidly become complex, and must be repeated for every set of data handled within the application. Factors influencing the complexity of the task are the interaction among the various data sets, the number of processors in the system, and the interconnection network of the processors. Should one of these parameters change, the performance of the application is in jeopardy.

Once the data has been distributed among processors, the programmer must arrange for the correct communication of intermediate values. Border, or end cases make this job particularly complex. Data distribution is a tedious task and as the size of the application and the number of data sets increase, it can become overwhelming.

Each of the components of data parallel application development mentioned above, the basic function applied to the data set, iteration over the data space, data distribution among processors, and communication of intermediate values, are common to all data parallel applications. The extensions we have added to the MPL are designed to free the programmer from concentrating on these mundane details of the data parallel style. Since we are removing the tedious burdens from the programmer, some other part of the system must become responsible for them. In our case, the compiler will recognize the keywords and constructs that comprise the extensions, and will generate code that manages iteration, synchronization, distribution, and communication for the programmer.

In our subsequent discussions, we will refer to the individual members of a data set as *elements*. The data set as a whole will be referred to as the *data parallel object*. We will begin our discussion of the data parallel mechanisms by explaining the nature of the data parallel class and the syntax used to define such a class in Section 3.1. We also discuss communication semantics in this section. In Section 3.2, we explain the syntax and semantics of the data parallel method types the language provides. Section 3.3 will deal with the creation and distribution of a data parallel object. We then turn to the additional tasks needed to support data parallel objects in Section 3.4. Finally, we will discuss the combination of our data parallel extensions with the original control parallel mechanisms of Mentat in Section 3.5.

3.1 Data Parallel Class Definition

The definition of a data parallel class is the primary means by which the programmer conveys information to our compiler about a data parallel computation. All data parallel applications are concerned with one or more data sets. Thus, it is natural to capture the data set representation and the data parallel operations for the data set within a single class. We have taken advantage of this fact in our extensions.

Data parallel computations are also characterized by the fact that every element is structurally identical to every other element of the data set[1]. The actual values of each element and their relative ordering within the set are the only distinguishing factors among the elements. The operations applied to a single element are identical to the operations applied to every other element of the data set. Thus, an entire data parallel data set and its operations can be defined by specifying the structure of a single element, and the operations applied to that element. Specifying a data parallel computation in terms of a single element

1. This characterization has been true in the past, however, the emergence of nested parallelism may cause a realignment of this view. For this thesis we will assume that all elements of a data set are structurally equivalent. Nested parallelism is one of the primary areas for expansion of the research described in this thesis. At that time, this assumption will be re-examined.

```
1:  dataparallel mentat class image{
2:          //private member variables which specify an element.
3:          int pixel;
4:          float another_pixel;
5:      public:
6:          // public member functions expressing the data parallel
7:          //     methods.
8:          void AGG scale ELEMENT (int value);
9:          void OVR overlay_pixel (RMAJ int *pixel_data);
10:         int RED min_elem(int curr_min,int curr_elem_mbr);
11: };
```

Figure 3.1 Data parallel class definition.

is the approach we have used in creating our data parallel extensions to the MPL. We call this approach *element-centered*. Fundamentally, this concept is not new to data parallel languages [2, 11, 14, 15, 18, 19]. However, we have extended the notion to encompass subset parallelism.

Figure 3.1 illustrates a simple example of a data parallel class and conveys the idea of an element-centered 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 new keywords indicate to the compiler that data parallel transformations must be applied to this 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. However, the method definitions differ from those of C++ classes. The member functions of a data parallel class are annotated to convey information to the compiler. In this section, we give brief descriptions of the member variables and member functions of the data parallel class and explain the semantics of data parallel operations.

3.1.1 Dataparallel Class Member Variables

The programmer should think of the data portion of the data parallel class definition as a template[2] for an element. The member variables of the class definition represent a single element of the data set. In our example, an element would have two components, an integer value (`pixel`) and a floating point value (`another_pixel`). These member variables are declared on lines 3 and 4. Any primitive or user defined type is allowable as a member variable of an element[3]. The effect of this type of data representation is that each

2. The term template in this context should not be confused with C++ templates.

element of the data set is essentially a structure whose members are the member variables declared in the data portion of the class definition.

The programmer does not specify the actual size of the data set in the data parallel mentat class definition. Also, the member variables are not defined as arrays unless the programmer is specifying a data set in which the elements themselves are arrays. When an instance of the class is declared, multiple elements will be created. The actual number of elements is specified at this time. Once created, the elements can be referred to individually, in subsets, or as a whole. We will discuss both object creation and element addressing later.

3.1.2 Dataparallel Class Member Functions

Our extensions include three types of member functions which are specifically designed as data parallel operations. These types are *overlay* operations, which are used to initialize the data parallel object, *aggregate* operations, which are applied to all elements of the data parallel object, and *reduction* operations, which allow the programmer to distill certain information from the values of the data set.

The semantics and communication characteristics are different for all three types. For example, aggregate and overlay operations straightforwardly iterate over each element subset applying an operation to each in turn. On the other hand, reduction operations automatically merge intermediate results of the computation on each element. Method annotations provided by the programmer serve to discriminate between operation types. In addition to distinguishing the method types, the annotations specify the subset of the data dealt with in the operation and determine the size of the return type.

All types of data parallel methods: aggregate, reduction, and overlay, may have local variables and arguments of primitive or user defined types[4]. 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 3.1, lines 8-10.

3. Allowing data parallel classes as member variables is one step in enabling nested data parallelism. Again, this topic is not considered in this thesis but is the subject of future research.

4. Data parallel objects as local variables would result in nested data parallelism.

3.1.3 Deterministic Data Parallel Semantics

Typically, when the communication semantics of a parallel language are discussed, communication occurs between two processors. In contrast, our language extensions attempt to shield the programmer from hardware specific details such as the number of processors or the interconnection network topology. Therefore, in the following discussion and in reference to our data parallel extensions, data parallel objects should be thought of as independent, interacting entities which communicate by exchanging the values of particular elements or subsets of elements.

There are three types of communication that occur with respect to data parallel objects. The first, *simple communication*, involves only the value of the element that is the target of the current method. *Local communication* implies that values of the data parallel object other than the current element are required. *Non-local communication* occurs when element values of a data parallel object other than the one upon which the method is invoked are required.

The member function annotations and the internal definition of the function body determine the type of communication of a method. A method uses simple communication if the only value used within the function body is that of the current element. A method falls into the local communication category if relative addressing mechanisms, explained in Section 3.4.3, are used within the function body to address neighbor elements of the current element. Finally, methods which have a data parallel argument fall into the non-local communication category.

Of the three types of data parallel methods, aggregate, reduction, and overlay, only aggregate methods may reference neighbor elements or accept a data parallel object as an argument. Therefore, aggregate methods are the only type of data parallel method to exhibit non-local communication. Overlay and reduction operations are restricted to simple communication since they can only access the current element. Figure 3.2 summarizes the communication possibilities for each method type.

An aggregate method is not limited to one category of communication, rather, any combination of simple, local, and non-local communication may occur within a data parallel method. However, when methods are classified according to their communication

Method Types	Communication Types		
	Simple	Local	Non-Local
Aggregate	✓	✓	✓
Overlay	✓	✗	✗
Reduction	✓	✗	✗

Figure 3.2 Communication capabilities by method type.

types, non-local communication has precedence over local communication which in turn has precedence over simple communication. For example, a method with non-local and simple communication would be classified as a non-local type of operation.

Simple communication is straightforward for the programmer to understand. The programmer simply uses the name of an element member variable to access the value of the current element. Local and non-local communication is more complex. Accessing the values of neighboring elements and elements of data parallel arguments requires an understanding of the pre-copy semantics of our data parallel extensions. Next, we present a discussion of pre-copy semantics in conjunction with our explanation of local and non-local communication.

The main characteristic of a data parallel computation is that the operation being applied to the data set is applied "logically simultaneously" to every element of the data set. This is the reason that this style of programming is considered parallel. Another way to understand this concept is to realize that each element is being operated upon concurrently. If there happen to be 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 elements. When multiple data are assigned to a single processor, the degree of parallelism decreases. This situation demands that the computation proceed as if there were as many processors as elements. Each processor must iterate through the data elements for which it is responsible, applying the operation to each in turn. Difficulties arise when the operation must appear as if it is occurring in parallel for all elements even though the implementation is actually sequential at each processor.

As an example, assume we have a processor which is responsible for five neighboring elements (a vector indexed from 0 to 4) of a data parallel object and the processor has been directed to execute `op1()`[5] which updates the value of each element. Also assume that when `op1()` is applied to an element, neighboring element values are used to compute the value assigned to the element. Before `op1()` has been applied to any of the elements, all elements have the same value *a*. The desired result of `op1()` is that all elements will have the value *b*. After processing the first element, `element0` has the new value *b* that was computed by `op1()` using the values (*a* and *a*) of its neighboring elements (in this case `element2` and `element4`). Therefore, our vector has the values <*b, a, a, a, a*>. Next, `op1()` is applied to `element1` with two possible outcomes.

The first outcome is the expected outcome of a sequential program, the second is the outcome according to the data parallel style where the operation is applied simultaneously to all elements. In the first case, the neighbors of `element1` are accessed, returning *b* and *a*. The result of `op1()` is then computed and is *c*. Thus our vector has the values <*b*, *c*, *a*, *a*, *a*> and is incorrect according to data parallel semantics.

In the second case, a copy is made of the vector before the iteration over the elements of the vector begins. All references to the elements are then resolved using this pre-copy of the elements and all updates are made to the current copy. Therefore, when `op1()` is applied to `element1`, the values of neighboring `element0` and `element2` are *a* and *a* respectively, not *b* and *a* because the values were retrieved from the pre-copy instead of the current copy. The result of `op1()` applied to `element1` is *b* as it was when `op1()` was applied to `element0`.

The pre-copy approach is employed in our data parallel extensions. While the actual mechanisms for achieving the desired results are discussed in Chapter 4, we address the implications of using this technique here. Pre-copies of data parallel object data are made whenever a data parallel operation is invoked. For local communication, the copy is made from the current copy of the data parallel object that is being invoked. For non-local communication, the copy is filled in with data communicated from the processor(s) at which the data parallel argument is being maintained. The programmer is not given explicit access to these pre-copies, i.e. they may only be modified by the compiler. Mappings are

5. The actual function of op1() could be as simple as addition.

maintained between the pre-copy and the current copy. Subsequently, references to the current copy are resolved from the pre-copy. Thus, local updates of an element are not visible to any other element in the data set until the operation is complete. At that point, the pre-copies are considered invalid, and the newly computed values are considered the current values of the data set.

3.2 Data Parallel Method Types

As mentioned previously, the more tedious tasks of data parallel method development are the construction of iteration control statements for a data set and communication of data values which are not local to a processor. In our approach to data parallel language design, the compiler manages many of these details automatically. We only require the programmer to annotate the method definition and specify the body of the function in an element-centered fashion. The annotations essentially extend the type of the function. Using this information, the compiler determines the values (other than the value of the current element) that are needed to complete the computation and infers the remaining control structures that are needed to complete the operation.

Recall that we distinguished three types of data parallel operations in Section 3.1.2. Function prototypes for all three types of methods are shown in the example of Figure 3.1. Figure 3.4 gives a brief grammar that defines the annotation syntax for each type. In the following sections, we will discuss the details of each method type. We start with a subset of **`AGG`** methods which we call *simple aggregate methods*. This method type illustrates many of the important concepts of data parallel objects, including the mechanics of writing a data parallel method body. We then turn to more complex aggregate types since they illustrate the various communication semantics. Finally, we handle reduction and overlay methods.

3.2.1 Simple Aggregate Methods

The `scale()` function prototype on line 8 of Figure 3.1 has been annotated to indicate the type of the data parallel operation and the size of the data portion that the method deals with. The first production of the `<dp_mbrfcn>` non-terminal in Figure 3.4

```
dataparallel mentat class image{
        //private member variables - “single element”.
        int pixel;
        float another_pixel;
    public:
        // public member functions - data parallel methods.
        void AGG scale ELEMENT (int value);
        void OVR overlay_pixel (RMAJ int *pixel_data);
        int RED min_elem();
};

void AGG image::scale ELEMENT (int value){
        pixel = pixel * value;
}

int RED image::min_elem(int curr_min,int curr_elem_mbr) {
    if (curr_min < curr_elem_mbr) return curr_min;
    else return curr_elem_mbr;
}

void OVR image::overlay_pixel(int RMAJ *value) {
    pixel = *value;
}

int *value;

my_image.scale(10);              /* Invocation on a matrix */
my_image[1].scale(10);           /* Invocation on a row */
my_image[4][4].scale(10);        /* Invocation on an element */
int x = my_image.min_elem(pixel);
my_image.overlay_pixel(value);
```

Figure 3.3 Data parallel class and method definition.

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`**.

An aggregate function is applied to every element (or subset) of the data set. The simplest kind of data parallel operation, of which `scale()` 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. The `<subset_specifier>` annotation (shown in the example production), in combination with the instructions of the actual function, allows the programmer to make this distinction.

In our example `scale()` method, the annotation is **`ELEMENT`**. This indicates that the method should be applied to each individual element of the data set. Alternative

Grammar:

<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 regular C++ argument...
<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 3.4 Grammar for data parallel object methods.

annotations, **`ROW`** or **`COL`**, specify the additional types of subsets that can become the target data of an aggregate operation.

There are two points to remember when developing a data parallel member function: the structure of the subset being operated upon and “location independent” references to data elements. First, the body of the function must be written as if the operation is being performed upon a single subset. This subset is indicated by the `<subset_specifier>` annotation. 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. Secondly, the body of the function must be written without reference to any <u>specific</u> element. This is what is meant by a “location independent” reference. In our `scale()` example, 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 in order to index into the data set properly. Thus, the function body must be devoid of any reference to a specific element or subset. 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. The relative addressing mechanisms will be discussed in Section 3.4.3.

These techniques, the `<subset_specifier>` annotation and the interior style of the member function, fit with our element-centered approach to data parallel class definition. The important point to note is that the word "element" in the term element-centered really means subset. The programmer must think in terms of a single abstract subset when developing an aggregate method[6]. If this is done correctly, the iteration required for applying the operation to each individual subset in the entire data set will be managed by the compiler without any further intervention from the programmer.

Invocation of an aggregate operation is shown on lines 27-29 of Figure 3.1. Assume that a data parallel object identified `my_image` has been declared as a two dimensional data parallel object of type `image`. Object declaration will be discussed further in Section 3.3. The invocation syntax is exactly the same as the C++ invocation of a member function. The compiler ensures that the iteration provided within the operation is restricted to the elements indicated by the invocation. Therefore, the invocation on line 27 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 28; 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 29. In this manner, the programmer is given a great deal of flexibility in invocation of data parallel methods on a particular data parallel object and the details of the iteration required to perform the desired effect are hidden within the annotations and the data parallel class definition.

A primitive or user defined type may serve as the return type of a data parallel class member function. Data parallel objects are not allowed as return values. Again, the notion of an element-centered 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. 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

6. Note that data parallel member functions may have different <subset_specifier> annotations. Thus, the "subset" may vary from method to method.

of actual results to be returned. For example, suppose that our `scale()` method returned an integer. Then the invocation on line 27 would cause space for *k* integers to be allocated, where $k = n \times m$, 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. The results computed for each element would be collected by the compiler and assigned to the corresponding positions of the return variable.

3.2.2 Complex Aggregate Methods

Our data parallel mechanisms also allow the programmer to specify a data parallel object 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. The syntax of an aggregate method argument is shown in Figure 3.4 as the production of the <*agg_arg*> non-terminal.

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. 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. The programmer would indicate this with a **`1x1`** `<combination_rule>` annotation[7]. 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 matrix multiply where every row of one matrix is combined with every column of another matrix.

7. This annotation implies that there are an equal number of subsets in each object.

3.2.3 Overlay Methods

Overlay operations provide for the initialization of the member variables of data parallel objects and require an **`OVR`** annotation (see the second production in Figure 3.4). Because this method type is meant primarily for initialization of data parallel objects, the return type is always void. The arguments to overlay operations may be any primitive or user defined type other than a data parallel type. The values passed via actual arguments are assigned in either a row-major or column-major order to the member variables of the data parallel object. This choice, row- or column-major, is indicated by the use of either **`RMAJ`** or **`CMAJ`** respectively as the annotation to the formal argument. For example, to initialize the `pixel` values of `my_image` with an integer vector of values, perhaps read in from a file, the `overlay_pixel()` method shown on lines 21-23 in Figure 3.1 would be invoked.

3.2.4 Reduction Methods

Reduction operations are global operations that reduce a set of values to a single value. These operations are tagged with a **`RED`** annotation. Reduction operations must be binary, commutative, and associative. Examples of reduction operations include minimum, maximum, and sum.

Standard reduction operations are often included in data parallel languages. Rather than define a set of supported reduction operations we have chosen to allow the programmer to define their own reduction operations. The compiler will then generate code to perform the required operations and necessary data movement. For example, consider the `min_elem` function of Figure 3.1. We recognize that the syntax for this method type is a bit non-intuitive. The reduction function must take two parameters, and return a value (or reference). The two parameters and the result must be of the same type. When the function is invoked the programmer specifies which member variable should be used within the operation by using it's name as the actual argument to the function. The result is that `min_elem` is applied iteratively to each element's `pixel` member and the result of the previous invocation.

3.3 Allocation and Initialization of Data Parallel Objects

3.3.1 Object Creation and Distribution

Arguments to new()

Data parallel objects must be explicitly created using the operator `new()`. In particular, the data parallel extensions to the MPL require the use of an overloaded `new()` operator. 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. Figure 3.5 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. If either of these arguments is given the value one, then the data parallel object is treated as a vector. If both arguments are assigned a positive value other than one, then the data parallel object is treated as a two dimensional array. If both arguments are specified as one, then the data parallel object is assumed to have only a single element. A zero or negative value for either argument is considered an error. The argument values may be expressions. Currently, our data parallel additions to the MPL support one and two dimensional arrays and single elements. Future work will explore the

```
dataparallel mentat class matrix{
        //private member variables that specify an
        //element.
    public:
        // public member functions expressing the
        // data parallel operations.
}

main()
{
        matrix *matrix_A, *matrix_B;
        matrix_A = new (8, 8, 4PT, 2, matrix_B, matrix_mult()) matrix ();
}
```

Figure 3.5 Data parallel object declaration and communication patterns.

feasibility of allowing the programmer to declare data parallel objects using a variety of structures such as a tree or an unstructured group of elements.

For the *local communication* category the third and fourth arguments enumerate the type of communication that will be dominant within the data parallel object. As explained in Section 3.1.3, local communication occurs in terms of the data rather than the processors. 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 (when **NONE** is the pattern specified, the radius value is 0, otherwise it must be a positive value). Examples of local communication patterns with a radius greater than one are shown in Figure 3.6.

The local pattern and radius arguments are the mechanism by which this information is communicated to the compiler. We recognize that a number of member functions may be defined for the data parallel class, and that all methods will not exhibit the same local communication pattern. Thus, in order to achieve the best performance, the programmer should indicate the pattern that will be used most often in the methods of the object.

We do not allow for the dynamic changing of the communication pattern in the current language implementation. This type of functionality can either increase or decrease

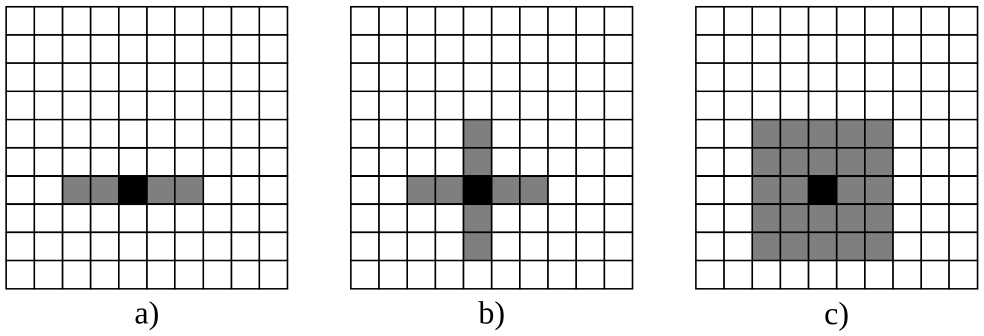

Figure 3.6 Three types of local communication pattern each with a radius of two.The darkly shaded element is the element to which the neighboring values are “communicated”. a) EW pattern. b) 4PT pattern. c) 8PT pattern.

```
void AGG matrix::mat_mult ROW (COL 1xN matrix);
matrix *grid_space1, *grid_space2;
grid_space1 = new (64, 64, 4PT, 2, grid_space2, mat_mult()) matrix();
```

Figure 3.7 Object Allocation and Distribution.

performance depending upon the skill with which it is employed. We feel that at the very least, the programmer should be aware of the dominant communications patterns within the computation and be able to communicate this information at compile time.

The arguments of the final category, *non-local communication*, provide information about interactions between the data parallel object being created and another data parallel object of the application. The sixth argument to the `new()` operation specifies the dominant function of the data parallel object being created. The fifth argument to the `new()` operation indicates the object which is most often used as the actual argument to the method specified by the sixth argument. Again, the programmer should be familiar enough with the application to specify the dominant non-local communication characteristics.

Allocation and Distribution

While the programmer supplies the actual arguments to the overloaded `new()` operator, the method is implemented by the compiler. The purpose of the `new()` operator is to allocate space for the object on the processors of the system. This is done using the information conveyed by the arguments to `new()`. Figure 3.7 shows a method prototype for a data parallel class `matrix`, the declaration of two instances of that class, `grid_space1` and `grid_space2`, and the creation of the data parallel object `grid_space1`. We will use this example to motivate our discussion of data parallel object allocation and distribution.

The dimension category of the arguments to `new()` indicates the base amount of memory that needs to be allocated for the data parallel object. In our example 64 * 64, or 4096 elements would be allocated. In some cases, the programmer will want to define border regions around the data set to handle edge conditions. Therefore, additional memory must be allocated for the border elements. The compiler performs an analysis of the data parallel class to determine the maximum local communication radius over all data parallel methods of the class. This number indicates the maximum width of any border which may be defined for the object. The maximum is added to the amount specified by the dimension arguments to get the total amount of data that must be allocated for the data parallel object. We will explain the mechanism for defining and manipulating border regions in Section 3.4.1.

Once the proper amount of space is determined, the compiler must establish a distribution pattern for the elements across the processors. Again, this is accomplished in the body of the overloaded `new()` operation. As explained in Section 3.2.1, the `<subset_specifier>` method annotations for the invoked object and the argument object shown on line 1 of Figure 3.7 indicate the subset of the corresponding object for which the programmer is responsible. In this case, the `<subset_specifiers>` are **`ROW`** and **`COL`** for the invoked object and the argument object respectively. However, in the context of data parallel object distribution, the invoked object and argument object `<subset_specifiers>` serve the additional purpose of expressing the optimal distribution of the two objects with respect to each other.

For an operation such as matrix multiplication, it is obvious that the proper distribution should place the rows of the invoked object and the columns of the argument object on the same processors. This distribution minimizes the communication needed to complete the operation for the object and thus results in the best performance. Exactly this type of information is conveyed by the method annotations and the arguments to the `new()` operator.

In our example, `grid_space1` is the invoked object, and `grid_space2` is the argument object. The arguments specified in the non-local communication category of the arguments to `new()` are an object name and a method name. The method `mat_mult()`, defined on the `matrix` class, will be invoked on the object `grid_space1` more often than any other method. Additionally, the object `grid_space2` will serve as the actual argument to the `mat_mult()` method for the greatest percentage of those invocations. The identification of the method `mat_mult()` indicates the subsets of the invoked and argument objects that will be used when the method is applied. In particular, these subsets will be referenced in conjunction with one another. Therefore, the subsets indicated by the method annotations are precisely the subsets which should be jointly distributed to the same processes.

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 hints 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 [21, 22]. Cooperating with the run-time system in this manner allows us to divorce ourselves from the underlying machine architecture.

We believe that the decomposition, distribution and alignment as described in Fortran-D and other data parallel languages (see Section 2.3) are equivalent to our mechanisms of specifying the communication patterns. These languages require the programmer to know 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.

3.3.2 Data Parallel Class Constructors

Constructors of data parallel mentat classes are analogous to constructors defined on regular C++ classes. The actions specified within the constructor are carried out after the object has been created and before any other methods are applied to the object. The programmer is responsible for defining the constructor, but we impose certain limits on the content of the constructor. The constructor can be thought of as a data parallel operation for which the annotations have default values. The method type is aggregate (**`AGG`**), the `<subset_specifier>` is **`ELEMENT`** and data parallel objects are not allowed as arguments. Because the method being defined is a constructor, no results may be returned. Also, data parallel methods defined on the class may not be invoked within the constructor. Therefore, the programmer must develop the body of the constructor in the same manner as the `scale()` example of Figure 3.1. This functionality allows the programmer to initialize the elements of the data parallel object to any default values they feel are necessary.

3.4 Data Parallel Object Support

3.4.1 Border Management

Consider a data parallel object which has been created as a two-dimensional matrix and which has an aggregate method using local communication defined on the class. The local communication pattern of the method is **4PT** with a radius of one. For the majority of elements internal to the data parallel object, the neighbor elements of the current element will themselves be elements of the data parallel object. However, Figure 3.8 illustrates the situation when a current element (the darkly shaded element in the figure) on the edge of the object must reference neighbor elements. The neighbor elements are not a part of the data parallel object. To prevent the neighbor element references from returning undefined values, we allow the programmer to define border regions for data parallel objects. Currently, the extensions allow three types of border management to be used for a data parallel object. These are *wrap-around*, *cyclic*, and *buffer*. The programmer specifies a border management policy by using one of the predefined border methods provided for every data parallel class. These methods may be invoked for a member variable of a data parallel object at the same scope as the object name, but not within a method of the data parallel class.

Figure 3.9 lists the form of the methods and gives a pictorial representation of the resulting border management for each. The wrap-around policy returns the element from the opposite border in the same row or column of the current element. Thus, if an element on the north border of the object references it’s north neighbor the value returned is that of the southern-most element of the same column. If a cyclic management is specified, the programmer must specify a direction for the cycle. The element returned is then the element

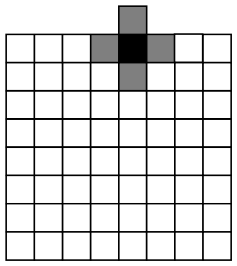
Figure 3.8 Reference pattern for an edge element.

Border Method Prototypes

```
obj.member_variable->border_wrap(<edge>);
obj.member_variable->border_cyclic(<edge>, <direction>);
obj.member_variable->border_buffer(<edge>, range, value, width);
```

Examples

```
A.pixel->border_wrap(N);
A.pixel->border_cyclic(N, E);
A.pixel->border_buffer(N, [-2:5], 0, 2);
A.pixel->border_buffer(E, [-2:5], 0, 2);
A.pixel->border_buffer(W, [-2:5], 0, 2);
A.pixel->border_buffer(S, [-2:5], 0, 2);
```

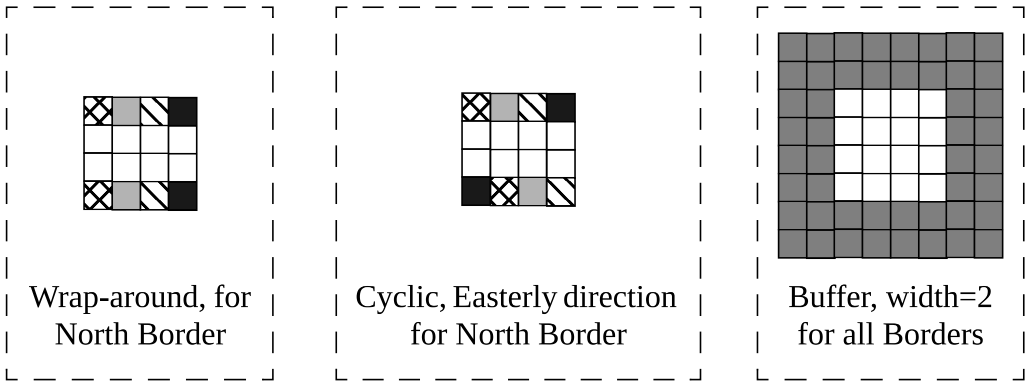
Wrap-around, for North Border
Cyclic, Easterly direction for North Border
Buffer, width=2 for all Borders

Figure 3.9 Border management methods.

from the opposite border in the next row or column in the direction specified. For example, a cyclic policy with an easterly direction defined for the north border would return the southern-most element of the column immediately to the right of the current column. Finally, if a buffer management is specified, then a padding of dummy elements are initialized to the value specified in the `border_buffer()` invocation. References to a north neighbor will simply return the value contained in the dummy element immediately above the current element. The range argument allows the programmer to assign different values to the elements within the same buffer.

The border management methods can be invoked for an object multiple times during the application. This allows the programmer a great deal of control over the edge conditions because a certain border policy is not tied to a particular data parallel method. If the programmer chooses not to specify a border management policy, then wrap-around is the default.

```
1:  void AGG image::sum_rows ROW ()
2:  {
3:     for (int i = 0; i < this.num_rows(); i++)
4:        this[0].pixel += this[i].pixel;
5:  }
```

Figure 3.10 Predefined method use.

3.4.2 Predefined Methods

Because the programmer must have access to information such as the number of rows or columns in an object, certain methods are predefined for all data parallel classes. These include `num_rows()`, `num_cols()`, and `num_elements()`. These integer functions will typically be invoked within data parallel operations as part of the programmer defined iteration, and return exactly what one would expect from their names. Figure 3.10 illustrates the use of a predefined method on line 3.

3.4.3 Relative Addressing Mechanisms

Relative Addressing

Within an aggregate member function, neighboring subsets of the current subset of the invoked object may be accessed in a read-only fashion. The relative addressing mechanisms are used for this purpose. Access to the neighboring subsets of an argument object using the relative addressing is not allowed. Subsets to the north, south, east, and west of the current subset are referenced using `N()`, `S()`, `E()`, and `W()` respectively. These functions are defined for all data parallel classes and return a pointer to an element. Thus, `S()->pixel` refers to the subset below the current subset in a two dimensional array.

Relative addresses may be chained together to form more complex reference patterns. Thus, `S()->W()` refers to the subset diagonally south and west from the current subset. We restrict relative access to static patterns, e.g., `N()->E()`, rather than dynamic patterns that are run-time dependent. By restricting ourselves to static patterns we can analyze the communication requirements of a member function at compile time. In addition to the NEWS methods, we also define `PRED()` and `SUCC()` for one-dimensional objects and subsets. Not all combinations of relative addresses are legal. Some illegal combinations can be detected at compile time, e.g., `N()` in a function that is annotated with a **`COL`** `<subset-specifier>.`

Subset Element Addressing

Within the subset specified by the <subset_specifier> annotation of a data parallel method, the programmer uses the standard array indexing notation to reference the elements of both invoked and argument objects. See Figure 3.10, line 4 for an example.

3.5 Integration of Control and Data Parallelism

For control parallel objects, the Mentat Run-Time System monitors the use of results of mentat object member function invocations. Data parallel objects are monitored in the same fashion. We demonstrate the combination of control and data parallelism in Figure 3.11. In this example, the results of the mentat object member functions are `x`, `y`, and `z`. The semantics allow the method calls on lines 16-18 to be executed in parallel. On line 19, the result of the data parallel method is used as an argument to a control parallel method. Furthermore, the entire code fragment itself could be a mentat object member function implementation executing concurrently with other mentat object member functions. Although this is not an actual application, this example clearly demonstrates that the data parallel extensions we have developed allow Mentat to support both types of parallelism. In addition, the simplicity of this integration is a strong argument for the design of our data parallel extensions.

```
1:  dataparallel mentat class data_parallel_obj {
2:         // private member variables
3:     public:
4:         // public member functions
5:         int AGG row_sums ROW ();
6:         ...
7:  }
8:
9:  ...
10: float x, z;
11: int y;
12:
13: control_parallel_obj A, B;
14: data_parallel_obj my_image;
15:
16: x = A.op1();
17: y = my_image.row_sums();
18: z = B.op1();
19: B.op2(y);
20: ...
```

Figure 3.11 Control and data parallel method invocations.

Chapter 4 Implementation Model and Translations

To be useful, programming languages require an implementation. The compiler must translate a source language program onto a target language program. In procedural languages such as C and Pascal, the target is an assembly language. Many details must be handled, such as the stack. The assembly language provides a call-return mechanism, however it is up to the compiler of the source language to implement a procedure stack using the mechanisms provided by the assembly language.

The data parallel Mentat programming language is no different. In control parallel Mentat, the translated source code handles recognizing which portions of the code can be executed in parallel. For our data parallel extensions the target language is the run-time library of the Mentat programming language. Among the details that need to be managed for the extensions are data distribution, recognition of what data must be communicated before an operation to ensure all of the data is local to the processor once the computation begins, and inserting the proper iteration type for the operation to proceed on the invoked object (or subset of an object). Therefore the target of a data parallel source is a control parallel code which synchronizes cooperating objects at the proper points.

Quinn [19] has demonstrated that a data parallel program can be converted to run efficiently on a MIMD machine by synchronizing at the proper points. Our extensions differ in that the programmer is allowed to operate on distinct elements of data parallel objects (versus all elements of a type), and may also operate on any subset of a data parallel object.

We do not intend to show that ours is the best implementation possible. Rather, we will demonstrate that an implementation exists. In almost every case the implementation is naive, there are optimizations that could be done which would greatly improve the performance and the elegance of the implementation.

The basic structure of the implementation is a master and a slave. The slave is a separate mentat object, the master is a C++ object. Between the master and the slaves, distribution, communication, and synchronization is handled for all data parallel objects declared in the source program. In the following sections we explain the implementation model and explain the code translations which are necessary to support the model.

4.1 Implementation Model

In many ways, the issues that we must resolve with respect to the implementation of our data parallel extensions are very similar to the issues that the designer of a SIMD architecture faces. We have chosen the master-slave paradigm as the basis of our implementation model for the data parallel extensions to the Mentat Programming Language. This same model is used in many SIMD architectures. Such architectures employ a host node, or master, which executes the application program and sends instructions to the processing elements, or slaves, each of which apply the instructions to a particular portion of the data. Because the host node is a single point of control, synchronization of the processing elements is straightforward. This model is well suited to producing semantics that are easily understandable by the programmer. While the master-slave paradigm of the SIMD architecture model is constructed in terms of host and node processing elements, our implementation depends on master and slave components which are designed as mentat objects (processes) assigned to particular processors which are connected in a MIMD fashion. The master process acts as the source of a single instruction stream, while the slaves manage the multiplicity of the data. Thus, data parallelism is mapped onto control parallelism.

Once we chose the master-slave paradigm, the remaining model design decision was to determine the delegation of data parallel objects to slave processes. Given that every data parallel object would be distributed across multiple processors, we faced two possibilities in designing the master-slave relationship. These are illustrated in Figure 4.1. In the first alternative, one slave per data parallel object is created on each processor. The second choice is to create one slave per processor which manages a portion of every data parallel object within the same address space. We chose to implement a single slave process per processor in order to minimize the amount of communication required between address spaces on a single processor. This type of communication will occur with a method exhibiting non-local communication patterns; i.e. references are made to the data parallel argument of the method. This paradigm also simplifies the management task of the master process because instructions only need to be sent to one process per processor instead of multiple processes per processor. Therefore, we create one slave process per processor, and each new data parallel object that is created is distributed among these slave processes. As

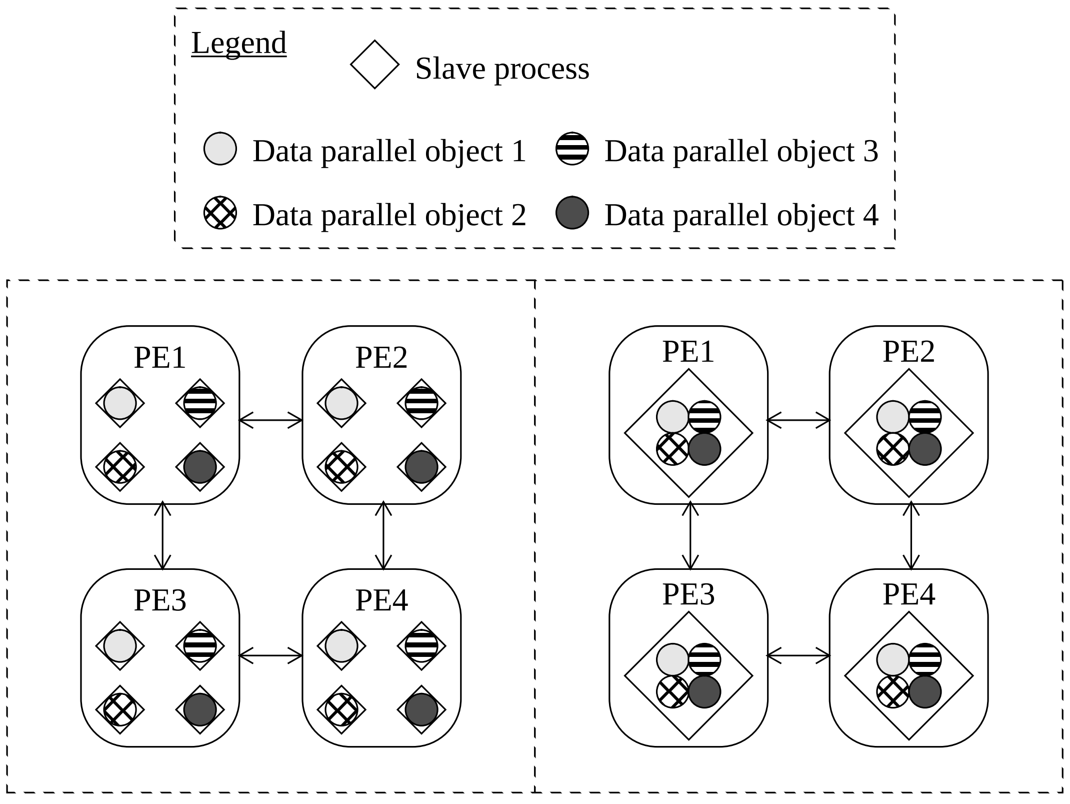

Figure 4.1 Example of two scenarios a) One data parallel object per slave process b) Multiple data parallel objects per slave process.

an example, consider four data parallel objects with twenty elements each and five processors. In this case, one master process and five slave processes would be created. Each slave would manage four elements of each data parallel object and the master process would transmit instructions to the five slaves. We will now present specifics about the implementation of both the master and the slave classes.

4.1.1 The Master

Figure 4.2 shows a pseudo-code definition of a master class. The master class is a C++ class which is defined by the compiler in the main program of the application. The master is responsible for creating and maintaining information about a slave process on each processor in the system. The master is an “intelligent” object which controls the action of the slaves. The slaves are essentially automatons that receive guidance from the master.

```
struct olist {
   // range of elements assigned to this slave process
   // list of methods defined on the data parallel class
   //     including the communication type and a place
   //     holder for return values
   // dimensions of the data parallel object
   // border information for the data parallel object
};

struct slist {
   MOname slave_handle;
   olist object_list;
};

class master {
      // member variables
      slist *slave_list;
   public:
      // member functions
      master() {// create a slave on each processor};

      // add a new object to object_list of each slave
      // partition overlay data and pass it to the slaves
      // determine communication pattern needed for a
      //     given operation and instruct slaves to
      //     exchange data
      // pass invocation of data parallel operation to the
      //     slaves
      // collect return results from slaves and return to
      //     the caller
}
```

Figure 4.2 Pseudo code for master class definition.

An instance of the master class is created by the compiler when the main function of the data parallel application comes into scope[1]. Since all control of the data parallel objects flows through the master, any creation or invocation on a data parallel object in the programmers source code is converted to an invocation on the master by the compiler. The master then performs the necessary tasks to ensure that the correct data is available at each slave for the ensuing computation, and oversees the invocation of the requested operation on every slave that is responsible for a portion of the data parallel object in question. Specifically, the duties of the master include:

1. Distributing newly created objects among the slaves.
2. Distributing overlay data among the slaves.
3. Directing the communication of data among the slaves.
4. Directing the invocation of data parallel operations by the slaves.
5. Collecting and consolidating partial results from the slaves.

1. Or when a mentat object is created which “contains” the data parallel object.

We will address each of these duties in turn.

Object Distribution. The master is responsible for distributing a newly created data parallel object among the slaves. When a new object is created, the master conveys the information concerning the dominant local and non-local communication patterns of the object to the run-time system. As explained in Section 3.3.1 this information is distilled from method annotations and arguments to the `new()` operator. Using this advice, the run-time system determines the appropriate decomposition of the object given the current system architecture, and returns it to the master. The master then distributes the object among the slaves located on each processor according to the directions from the run-time system. Note that this mechanism does not require a portion of the data parallel object be assigned to every processor, nor does it require equal portions of the data to be allocated to each participating slave. This mechanism enables the exploitation of a heterogeneous system architecture as well as the ability to adapt to a unevenly loaded system. It is the responsibility of the master to recognize when the data has been unevenly distributed and ensure that communication of data is properly managed in this case. The research to develop the run-time side of this mechanism is currently being conducted within the Mentat research group [21, 22].

Overlay Distribution. Administering the invocation of overlay operations is the second responsibility of the master. Upon invocation of an overlay operation on a data parallel object, the master receives the data to be assigned to the elements of the data set. The master possesses information about which data elements are located at which slave. The master uses this information to split the incoming overlay data into the appropriate pieces and forwards them to the corresponding slaves.

Communication between slaves. Directing the communication of data among the slaves is the most complicated task of the master. When an invocation of an aggregate operation requiring either local or non-local communication occurs, the master initiates the communication of data between slaves (and thus processors) before the invocation of the operation proceeds. This ensures that once the operation is underway, all necessary data is local to each processor, and that race conditions on updated element values will not occur during the execution of the operation. The actions of the master vary depending upon the

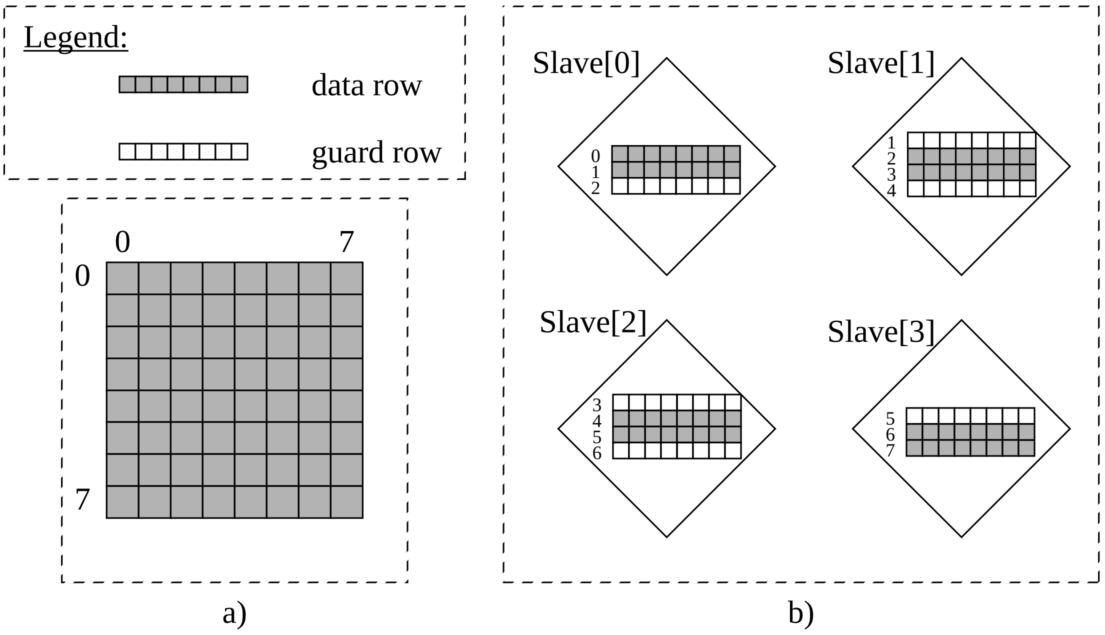

Figure 4.3 a) An undistributed data parallel object (abstract representation) b) Distributed data parallel object with guard regions at each processor.

type of communication required: local or non-local. We discuss local communication first followed by non-local communication.

Local communication requires that there be “guard regions” for the data held by a particular slave. These guard regions are place holders for any data which is not maintained at that slave that may be referenced during an operation. Figure 4.3 illustrates a data parallel object that is distributed by rows among four slave processes, and the guard regions that each slave maintains to ensure that there is space for all needed data. The shaded data rows indicate actual data assigned to a slave. The unshaded guard rows are comprised of actual elements and contain copies of data elements that are maintained by neighboring slave processes.

At compile-time, every method of a data parallel class is analyzed to discover it’s local communication pattern. Every data parallel method that employs local communication must use the relative addressing mechanisms defined in Section 3.4.3. The use of these mechanisms uncovers any local communication pattern to the compiler. The aggregate method in Figure 4.4 exhibits a local communication pattern. The reader will recall that possible communication patterns are `NONE`, `PRED`, `SUCC`, `NS`, `EW`, `4PT`, and `8PT`. Each pattern must have an associated radius. The communication pattern of the example

```
1:  void AGG image::stencil_ave ELEMENT ()
2:  {
3:     pixel = (N()->pixel + E()->pixel + W()->pixel + S()->pixel) +
4:               N()->N()->pixel + E()->E()->pixel +
5:               W()->W()->pixel + S()->S()->pixel)) / 4;
6:  }
```

Figure 4.4 Aggregate with local communication. Communication pattern is detected by the use of the relative addressing mechanisms.

`stencil_ave()` operation is `4PT` with a radius of two. The pattern type, `4PT`, is determined by the actual directions referenced. In this case, since all four are referenced, the pattern type is `4PT`. The radius is determined by the number of occurrences of the same direction in a sequence of references. In our example `N()->N()->`[2] indicates that the radius should be two. The radius value indicates to the master how many guard rows (or columns) must be exchanged before that operation may proceed, the pattern indicates on which border the guard rows should be located. (More precisely, the pattern specifies the neighboring slave from which the necessary data must be obtained.)

Each operation may have different space requirements for guard regions. This is determined by the radius portion of the communication pattern of the method. In order to avoid reallocating the guard and data regions before each operation, the compiler makes note of the largest radius of all communication patterns used in the methods of a data parallel class definition. When the object is created, enough guard rows are allocated to handle the maximum requirement of the object, even though they may not be used on every method invocation.

Recall that a data parallel operation that employs non-local communication must have a data parallel argument. Also recall that data parallel objects are distributed in order to minimize the data communication required for the method that is invoked most often (See Section 3.3.1). Therefore, when a method requiring non-local communication is invoked on a data parallel object, one of two situations must be in effect. Either the invoked object and the argument object were distributed to facilitate their interaction, or they were not.

2. E()->E()-> (or any of the other relative references in lines 4 and 5) also indicates a radius of two. Note that a reference sequence such as E()->N()-> would have radius one, while E()->N()->N()-> would have radius two.

In the first case, the data subsets of the invoked object and the argument object that must interact during the operation are locally maintained at each processor. The master does not need to arrange for any pre-communication of data between the slaves in order for the operation to commence. In the second case, it is possible that the appropriate data are not co-located at the processor. Therefore, the master arranges for copies of the needed data from the argument object to be communicated among the slaves. In order to accommodate such transfers of data, enough space to store the entire argument object must be allocated at every slave. This space must be allocated for every object since every object has the potential to be an argument to a data parallel method. This approach is problematic, since it seriously limits the problem sizes of applications using our extensions. However, it is not unique, and has previously been used by Hatcher and Quinn [11] to ensure proper access to needed data.

A solution to this problem entails the support of dynamic redistribution of objects. Such support would eliminate the need for allocating enough space for entire objects at every processor. By allowing a redistribution of the objects prior to method invocations, the correct data could be placed at the slaves without unnecessarily allocating enough space for the entire object. We believe that our current design is amenable to inclusion of a dynamic redistribution feature, however, it is beyond the scope of this thesis to implement such a feature. Additionally, we believe this shortcoming of the implementation does not affect the elegance of the language.

Directing Operation Invocation. The fourth task of the master, directing the invocation of data parallel operations, is straightforward. The master simply invokes the operation on each of the slaves. The slaves then proceed to asynchronously compute their results.

Collecting Results. If a particular method generates a return result, the master is responsible for collecting, organizing, and returning these results to the point of invocation. Each slave forwards its individual result to the master upon completion of their portion of the operation. The master then organizes the results in the proper order that is expected by the caller and returns the aggregate result.

4.1.2 The Slave

We now describe the functionality of the slave class. Figure 4.5 illustrates the pseudo code of a typical slave class. As with the master class, the slave class is generated by the compiler. However, the slave class is defined as a `sequential persistent mentat` class (line 9). The reader will recall from Section 2.2 that a `sequential persistent mentat` class guarantees that all invocations by object A on object B are received at B in the same order as their invocation at A. Instances of the slave class are distinct processes which communicate with the master class via member function invocation. Data parallel objects are decomposed and portions are distributed to the slaves by the master. Thus, each slave may be responsible for managing a portion of a number of data parallel objects.

Each slave will maintain a list of the objects for which it is responsible (line 11). For each of these objects, the members of the structure `solist` (lines 1-7) describe the state of an object at a particular slave. The primary copy (line 2) of an object represents the current values of the data elements before an operation. During a data parallel operation, these values are modified as directed in the method. The second copy (line 4) of the data is the mechanism used to prevent race conditions on the element values of the invoked object, and is the implementation of the pre-copy approach discussed in Section 3.1.3. Again, the values of the second copy reflect the state of the invoked object before the operation begins.

```
1:   struct solist {
2:          // primary copy of the data of the object maintained by
3:          //     this slave.
4:          // second copy of the data - used to prevent race conditions
5:          //     on local communication.
6:          // return value place holder for each method of the object.
7:   }
8:
9:   sequential persistent mentat class slave {
10:         // member variables
11:         solist slave_object_list;
12:     public:
13:         // member functions
14:         // marshall and send the data of a data parallel object to
15:         //     a peer slave using instructions from the master.
16:         // add received data to guard regions of a data parallel
17:         //     object
18:         // invoke a data parallel method on locally maintained data
19:         // return results to the master
20:         // two-copy the values of a data parallel object
21:  }
```

Figure 4.5 Pseudo-code for slave class definition.

Data parallel operations at the slaves are conducted by iterating over the local data set of the invoked object; thus they are operated upon sequentially as opposed to concurrently. We will refer to the element corresponding to the current index of the iteration as the “current” element and all others as “neighbor” elements. Elements of argument objects are referred to as “argument” elements. Updates of the current element value are recorded in the primary copy of the data, and are visible as long as that element remains the current element. Updates of neighbor and argument elements are not allowed by the language. During an operation, a read request may reference the current element, a neighbor element, or an argument element. Because element values of an argument object cannot be updated during a data parallel operation, requests for these values are filled from the primary copy of the argument object i.e. no pre-copy of the object is made, the primary copy is simply restricted to read-only access. Requests for the current element are met using the value recorded in the primary copy of the invoked object. This allows the programmer to compute intermediate results for the current element that are visible as long as the current element is in scope. However, as soon as an operation is complete for a current element, its status reverts to that of a neighbor element. Values for neighbor elements are supplied using the second copy of the invoked object. Thus, the values of neighbor elements are constant for every current element while in the scope of the data parallel operation.

In addition to the primary and second copies of the element data of each object, the slaves maintain a place holder (line 6) for the return values of every non-void data parallel operation defined for each object. These place holders are used to collect the result of the invoked operation on each element until every element possessed by the slave has been processed. These results are then collectively returned to the master. The master then collates the results and returns them to the caller of the data parallel method.

We proceed to the description of slave functionality. Each type of data parallel method, **`OVR`**, **`RED`**, and **`AGG`**, requires a slightly different sequence of events to occur between the master and the slaves.

Overlay Operations. First, upon invocation of an overlay operation, the slaves receive the new data from the master as a list of values. Each data point sent from the master represents the new value of a member variable of an element. The master has organized the values in the proper order for each slave, so the slave can simply iterate over the elements of the object assigning the new values to the member variable of the elements. Since there can be no local or non-local communication in an overlay operation, the new values are assigned directly to the primary copy of the object. After the overlay operation has completed but before executing the next instructions from the master, the slave performs a transfer of the primary data to the second copy. This operation can proceed concurrently while the master is preparing the next operation. The reason for making the copy at this time is performance. Subsequent operations which require local communication will not be slowed by waiting for the second copy to be made.

Reduction Operations. Reduction operations are applied to the elements of the data parallel object as described in Section 3.2.4. Each slave therefore produces a local result that is the reduction over the elements that it maintains. Each slave returns their result to the master who then performs the final reduction over all slave results. Recall, reduction operations must be binary, associative, and commutative. Since updates are not allowed as part of a reduction, element values are read directly from the primary copy of the object during the operation.

Aggregate Operations. Aggregate operations are the most complex of the three types. They require greater cooperation between the master and the slaves than overlay or reduction operations. Figure 4.6 illustrates the interactions of the master and slaves each time an aggregate operation is invoked. Upon receiving an invocation of an aggregate operation, the master determines if local or non-local communication is required by the operation. If local communication is required, then the master assembles instructions for the slaves to exchange data from the invoked object. If non-local communication is involved, the master assembles instructions for each slave regarding this exchange as well. Finally, the object upon which the operation is to be invoked is indicated in the instructions. These instructions are then passed to the slaves (message 1). The slaves marshall and send the data indicated in the communication instructions and wait for the receipt of similar messages from their counterparts (message 2). Once the data has been exchanged, the slaves record the values in the appropriate variables. At this point each slave will accept an invocation from the master (message 3). This invocation specifies which data parallel operation should be applied to the elements of the particular data parallel object. After computing their local results, the slaves return these values to the master (message 4). The

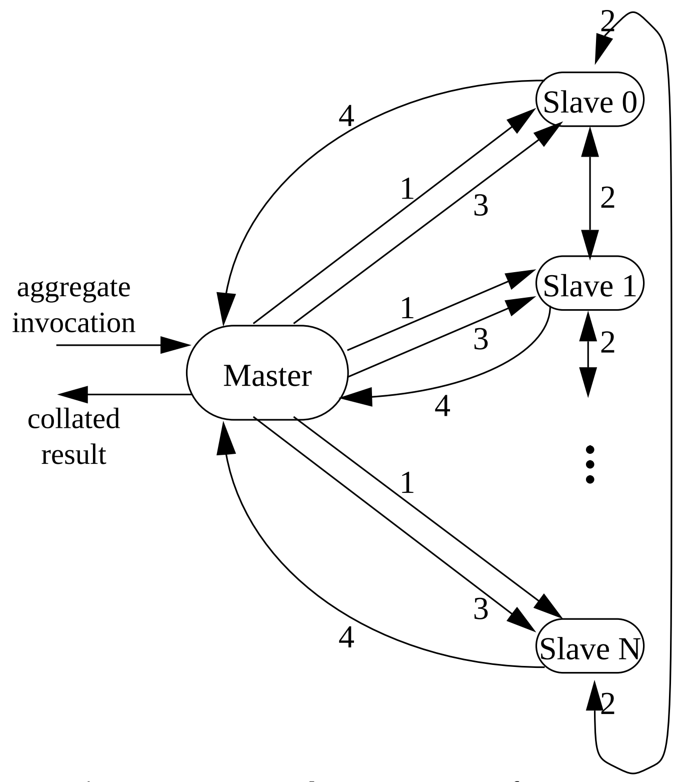

Figure 4.6 Master/slave interactions for aggregate operations.

master collates and returns the results to the caller. Since aggregate methods will typically modify the current element of the operation, the primary data of the invoked object will be transferred to the second copy after the completion of the operation for all elements. Again, this copy is done at the end of the operation to save time at the beginning of the next operation invoked on the object.

4.2 Translations

The annotations and communication characteristics upon which our data parallel extensions are based delineate four classes of data parallel methods. These classes may be ordered in terms of their increasing translation complexity. We have isolated eight types of transformation which are required to convert a programmer defined data parallel method into a working data parallel operation. As the complexity class of the method increases, so does the number of transformations which must be applied to generate the master/slave model we described in the preceding section. Currently, applications developed using these data parallel extensions are hand translated into the master and slave classes. Actual compiler support is a topic for future consideration.

In the remainder of this section, we first define the complexity classes and identify what translations are needed for each class. Then, to explain the translations, we present an example from each complexity class and the translations associated with that class.

4.2.1 Method Complexity Classes

The method complexity classes, in order of increasing complexity, are:

1. *Type 1*: Overlay, Reduction, and Simple aggregate methods with an ELEMENT <subset_specifier> annotation.
2. *Type 2*: Simple aggregate methods with a ROW or COLUMN <subset_specifier> annotation.
3. *Type 3*: Local aggregate methods.
4. *Type 4*: Non-local aggregate methods.

The distinguishing characteristic of type 1 methods is the <subset_specifier> assigned to the method. The subset for every method in this class is ELEMENT. These types of methods represent the concept of data parallelism most closely because each individual element of the object is operated upon “concurrently”[3]. For methods in this class, values of other elements of the invoked object, or any other object within the application, are not required to complete the computation for any one element.

A type 2 operation is any simple aggregate method for which the specified subset is ROW or COL. Recall that an annotation indicates the subset of the data parallel object for which the programmer is assuming responsibility. For example, a ROW annotation implies that the programmer intends to access multiple elements of a row as opposed to a single element[4]. Therefore, for performance reasons, the subsets must be organized so that no subset is divided across processors. If the initial distribution of the invoked object is not equivalent to the distribution mandated by the invoked operation, then a redistribution of the data may be required before the computation can proceed. Therefore, type 2 operations

3. Again, because of physical limitations such as the number of processors in the system the operations only appear to occur concurrently. Realistically, a number of the operations occur sequentially.
4. However, there are no references to the elements of a neighboring subset.

require all of the transformations applied to type 1 operations in addition to transformations designed to handle the redistribution of the invoked object.

All methods which make use of local communication as defined in Section 3.1.3 are grouped into the type 3 class of data parallel methods and are referred to as local aggregate methods. These methods may be annotated with all possible <subset_specifier>. As with type 2 methods, any type 3 method with a ROW or COL annotation may require a redistribution of the object before the operation proceeds. The additional complexity of this class arises from the fact that subsets other than the “current” subset are referenced by the programmer in the body of the function using the relative addressing mechanisms described in Section 3.4.3. Thus, those subsets must be locally available, in read-only form, at the same processor as the “current” subset. This requires a transformation which enables the communication of these values in addition to the transformations required for type 1 and type 2 methods. Also, the relative addresses must be translated to actual addresses.

The final class of methods are those designated as type 4 methods and referred to as non-local aggregate methods. These methods take a second data parallel object as an argument. As described in Section 3.1.3, references to values within the argument object are referred to as non-local communication. The transformations applied to type 2 and type 3 methods are aimed at the distribution of the data parallel *object* upon which an operation has been invoked. These same transformations must be applied to the data parallel object named as the *argument* to a type 3 method. Because there are two data parallel objects involved in non-local computations, there are two groups of subsets dealt with in the body of the method. The interaction among the subsets of the two objects is specified by the <combination_rule> annotation. This interaction is the differentiating factor for all type 4 methods and requires a new transformation in addition to the transformations applied to all other method types. The additional transformation handles the proper nesting and placement of iteration control structures for the object and argument.

4.2.2 Required Translations

The classification of data parallel methods into the various complexity classes can be done by evaluating the reference patterns within the methods, the method annotations, and any data parallel arguments to the method. This task is handled by the compiler.

Therefore, the programmer does not need to be aware of the particular class of a method; instead, the compiler detects the method class and applies the appropriate transformations automatically. The eight transformations alluded to in the previous paragraphs are:

1. *Iteration*. Provision of iteration control over the subsets of the invoked object as specified by the <subset_specifier>.
2. *Indexing*. Insertion of required array indices to all data parallel object references.
3. *Result Cardinality.* Allocation of space for the result set of each slave and for the operation as a whole.
4. *Object Distribution*. Coordination of the current distribution of the data parallel object with the distribution expected for the method.
5. *Guard Regions*. Allocation and management of space for guard regions of a data parallel object.
6. *Relative Address Conversion*. Conversion of relative addressing mechanisms to actual addresses within a data parallel object.
7. *Argument Distribution*. Coordination of the current distribution of the data parallel argument with the distribution expected for the method.
8. *Loop nesting*. Proper nesting of iteration control structures to handle combination of object and argument subsets.

Figure 4.7 illustrates the correspondence between each class of methods and the transformations. In order to explain each transformation in detail, we will provide an example of each method type. We will treat the four method classes in order of increasing complexity, and therefore at each level will only discuss the new transformations required for that particular type. A number of the transformations are parameterized by information

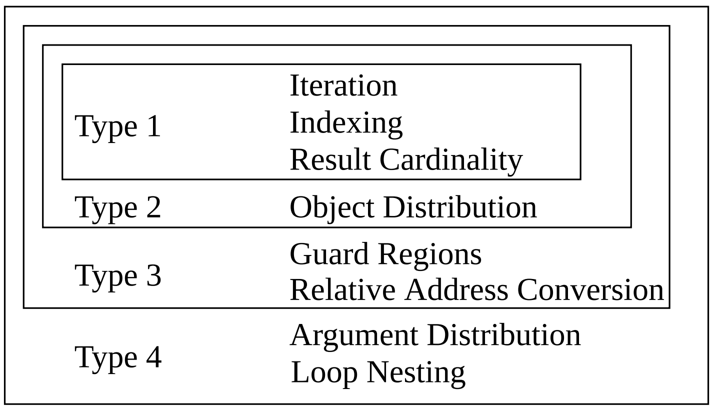

Figure 4.7 Class types and translations required.

```
void master::class_op1()
{
    // communication - redistribution and guard regions.
    // operation invocation.
    // return results to caller.
    // communication - redistribution.
}
```

a)

```
void slave::class_op1()
{
    // allocate result space
    // iterate over subsets (object and argument)
    //     programmer defined method body
    // return results to master
    // two_copy (see Section 3.1.3)
}
```

b)

Figure 4.8 a) Generated master class method. b) Generated slave class method.

that can only be known at run-time. Additionally, certain transformations are conditionally applied given certain run-time information.

Before proceeding with the actual examples, a discussion of the general form of the methods resulting from the transformations is in order. Each method defined by the programmer is converted into two new methods. One of these methods is defined within the master class, and one is defined within the slave class. Figure 4.8 illustrates the general structure of these two generated methods. Additional methods are defined on the slave class to handle the various communication patterns needed to ensure that all data are local to the appropriate slave (processor) before a computation begins[5]. These additional methods are invoked by the master during the communication phase of the generated master class method. We now turn to the method class translations.

Simple aggregate (ELEMENT), Overlay, and Reduction methods: Figure 4.9 illustrates a simple aggregate (ELEMENT) operation which returns an integer result. The translations required for this method type are translations 1 through 3. The portions of the master and slave class methods generated using transformations 1 through 3 appear in boldface. The same translations apply to overlay and reduction operations.

5. These communication methods are defined in a base class as was done in [13].

Original Data Parallel Method Definition and Invocation
Simple Aggregate (ELEMENT)

```
1:  int AGG image::scale ELEMENT (int value)
2:  {
3:     int result;
4:     result = pixel * value;
5:     return(result);
6:  }
7:  image pic;
8:  pic = new ()image();
9:  values = pic.scale();
```

Master Class Method Definition

```
10: int *master::image_scale(handle dpob_id, int value)
11: {
12:    // initial communication unnecessary.
13:    // allocate an int pointer to hold the results of each slave.
14:    for (int dp_i = 0; dp_i < num_slaves(); dp_i++)
15:       result_set[dp_i] = slave[dp_i].image_scale(dpob_id, value);
16:    // combine result sets into a single set.
17:    // post communication unnecessary.
18:    // return single result set to caller.
19: }
```

Slave Class Method Definition

```
20: int *slave::image_scale(handle dpob_id, int value)
21: {
22:    int result;
23:    // allocation of result space for each element.
24:    for (int dp_i = 0; dp_i < dpob_id.num_rows_held(); dp_i++)
25:       for (int dp_j = 0; dp_j < dpob_id.num_cols_held(); dp_j++) {
26:          result = dpob_id[dp_i][dp_j].pixel * value;
27:          result_set[dp_i][dp_j] = result;
28:    // return results to master.
29:    // pre-copy new values.
30: }
```

Figure 4.9 Original, master and slave method definitions for `scale()`.

The first translation, *iteration*, makes use of the invocation and the <subset_specifier> in order to provide the iteration bounds on lines 24 and 25 of the slave method `image_scale`(), and to control the invocations by the master on lines 14 and 15. Recall that a method may be invoked on a single element, a subset of elements, or all elements of a data parallel object. Therefore, the invocation indicates to the master and the slave which elements should be the subject of the operation. Our example invocation on line 9 involves all elements of the object `pic`. Notice that line 4 and line 26 are similar except for the array indexing. We will refer to line 4 as the method body and line 26 as the translated version of the method body. The `<subset_specifier>` is the mechanism by which the compiler determines it's responsibility for iteration over the subsets of the object. In our example, the `<subset_specifier>` **`ELEMENT`** indicates that all elements managed by every slave should be operated upon. Therefore, a doubly nested loop is inserted around the translated version of the method body[6].

The second transformation, *indexing*, makes use of information conveyed by the `<subset_specifier>` as well. Because a doubly nested loop has been inserted by the compiler, all data parallel object member variables referenced within the method body must be indexed to provide iteration control. In our example (line 26), two index variables have been added to all references to the member variable `pixel`. In the case of a **`ROW`** annotation, only one index variable would be added since the programmer would presumably provide the second one if needed. For example, `pixel[k]` in the method body would become `pixel[i][k]` in the translated version. Likewise, a **`COL`** annotation and `pixel[k]` reference would result in `pixel[k][i]`.

Finally, the third translation, *result cardinality*, depends upon information from the object `<subset_specifier>` and the invocation. Essentially, the number of subsets specified by the `<subset_specifier>` for the portion of the object indicated by the invocation is equal to the number of results that must be returned by the master. The slaves figure the result cardinality for their portion of the object similarly. This information is used to allocate space to hold the results in both the master and the slaves (lines 13 and 23 respectively). Code initiating the return of the results is inserted on lines 16 and 18 of the master class method and line 28 of the slave class method. This third translation is slightly more complicated for non-local aggregate methods. We will delay explanation of the translation for non-local aggregate methods for now.

Simple aggregate (ROW or COL) methods: Figure 4.10 illustrates a simple aggregate (**`ROW`** or **`COL`**) method which returns an integer result. In addition to translations 1 through 3, this type of method requires translation 4, *object distribution*. As before, the

6. All objects are assumed by the compiler to be two dimensional arrays at compile-time. If at run-time this is not the case, then `num_rows_held()` and/or `num_cols_held()` will return a value of one as appropriate. Thus the compiler allows for all possible configurations of the data parallel object.

Original Data Parallel Method Definition and Invocation
Simple Aggregate (ROW or COL)

```
int AGG matrix::row_sums ROW()
{
   int result;
   for (int j = 0; j < this.num_cols(); j++)
      result += this[j].value;
   return(result);
}
matrix A;
A = new ()matrix();
sums = A.row_sums();
```

Master Class Method Definition

```
int * master::matrix_row_sums(handle dpob_id)
{
   // if current distribution of object is not ROW
   //     then direct slaves to redistribute object by rows.
   // allocate an int pointer to hold the results of each slave.
   for (int dp_i = 0; dp_i < num_slaves(); dp_i++)
      result_set[dp_i] = slave[dp_i].matrix_row_sums(dpob_id);
   // if element values were modified and dominant distribution
   // is not ROW
   //     then direct slaves to redistribute object according
   //     to dominant distribution.
   // combine result sets into a single set.
   // return single result set to caller.
}
```

Slave Class Method Definition

```
int * slave::matrix_row_sums(handle dpob_id)
{
   int result;
   // allocation of result space for each element.
   for (int dp_i = 0; dp_i < dpob_id.num_rows_held(); dp_i++)
      for (int j = 0; j < dpob_id.num_cols_held(); j++) {
         result += dpob_id[dp_i][j].value;
         result_set[dp_i][j] = result;
   }
   // return results to master.
   // pre-copy new values.
}
```

Figure 4.10 Original, master and slave method definitions for `row_sums()`.

portions of the master class generated using this transformation appear in boldface. For this type of method, the <subset_specifier> of the method and the dominant distribution of the object must be compared. If they are not the same, then the data of the object must be redistributed to avoid the cost of off-host communication during the application of the operation. The master accomplishes this task by invoking inherited methods defined in the slave class. These methods are a collection of operations which handle the marshalling and sending of specific object subsets to a peer slave. Lines 13 and 14 of our example show that this redistribution takes place prior to the invocation of the data parallel method on the slaves. The semantics of the data parallel operations require that the processor managing the elements must maintain their values locally. Therefore, if elements of the data parallel object have been modified during the operation, it is necessary to undo the effects of the object redistribution. The master again oversees this action in the same manner as the initial redistribution as shown on Lines 18-21 of our example.

Local Aggregate methods: Local aggregate methods require the fifth and sixth translations, *guard regions* and *relative address conversion* respectively, in addition to the previous four translations. Both translations make use of information conveyed by the use of relative addressing mechanisms in the body of the programmer defined method. Figure 4.11 illustrates a local aggregate method using these mechanisms and the master and slave class methods which are generated by applying the transformations. For an operation of this type, the master is responsible for directing the population of slave guard regions prior to the method invocation. The compiler determines the amount of data needed on a per method basis by analyzing the use of the relative addressing mechanisms. This analysis, discussed in Section 3.1.3, returns a radius which indicates how many rows or columns in a certain direction are needed. In our example, a single guard row or column in each direction is sufficient. The master uses the compiler supplied methods mentioned earlier to achieve the transfer of the proper data regions. The pseudo-code for the *guard regions* translation is shown on lines 11 and 12. As with the *object distribution* translation, this transfer takes place before the invocation of the method.

The relative addressing mechanisms also direct the compiler to generate references to actual data elements in the body of the slave class method. The actual references generated comprise the sixth translation, *relative address conversion*. Since the needed data will be available within the same data structure as the elements which are the target of the operation, the `N()->`, `S()->,` etc. relative addresses are converted into expressions. These

Original Data Parallel Method Definition and Invocation
Local Aggregate

```
1:   void AGG image::stencil_ave ELEMENT()
2:   {
3:       pixel = (N()->pixel + E()->pixel + W()->pixel + S()->pixel) / 4;
4:   }
5:   image pic;
6:   pic = new ()image();
7:   pic.stencil_ave();
```

Master Class Method Definition

```
8:   void master::image_stencil_ave(handle dpob_id)
9:   {
10:      // redistribution unnecessary since annotation is ELEMENT
11:      // if guard regions are needed for invoked object
12:      //     then transfer regions among slaves.
13:      // result allocation unnecessary.
14:      for (int dp_i = 0; dp_i < num_slaves(); dp_i++)
15:          slave[dp_i].image_stencil_ave(dpob_id);
16:      // result combination unnecessary for this example.
17:      // return of results unnecessary for this example.
18:      // redistribution unnecessary for this example.
19:  }
```

Slave Class Method Definition

```
20:  void slave::image_stencil_ave(handle dpob_id)
21:  {
22:      // result allocation unnecessary.
23:      for (int dp_i = 0; dp_i < dpob_id.num_rows_held(); dp_i++)
24:          for (int dp_j = 0; dp_j < dpob_id.num_cols_held(); dp_j++)
25:              this[dp_i][dp_j].pixel =   dpob_id[dp_i-1][dp_j].pixel +
26:                                         dpob_id[dp_i][dp_j+1].pixel +
27:                                         dpob_id[dp_i][dp_j-1].pixel +
28:                                         dpob_id[dp_i+1][dp_j].pixel)
29:                                         / 4;
30:      // return of results unnecessary.
31:      // two-copy new values.
32:  }
```

Figure 4.11 Original, master and slave method definitions for `stencil_ave()`.

expressions are used to index the array data structure of the data parallel object as shown on lines 25-28.

Non-local aggregate methods: An example of the final data parallel method complexity class is shown in Figure 4.12. This method implements matrix multiplication for two data parallel objects. As before, all previous translations described are applied to

Original Data Parallel Method Definition and Invocation
Non-Local Aggregate

```
int AGG matrix::mat_mul ROW(COL 1xN matrix B)
{
   int result;
   for (int j = 0; j < this.num_cols(); j++)
      result += this[j].value * B[j].value;
   return (result);
}
matrix A, B;
A = new () matrix();
B = new () matrix();
A.mat_mul(B);
```

Master Class Method Definition

```
int * master::matrix_mat_mul(handle dpob_id, matrix B)
{
   // if current distribution of object is not ROW
   //     then direct slaves to redistribute object by rows.
   // guard regions unnecessary.
   // if current distribution of argument is not COL
   //     then direct slaves to redistribute argument by columns.
   // allocate an int pointer to hold the results of each slave.
   for (int dp_i = 0; dp_i < num_slaves(); dp_i++)
      slave[dp_i].matrix_mat_mul(dpob_id, B);
   // combine result sets into a single set.
   // return single result set to caller.
   // redistribution unnecessary.
}
```

Slave Class Method Definition

```
int * slave::matrix_mat_mul(handle dpob_id, matrix B)
{
   int result;
   // allocation of result space for each element.
   for (int obj_i = 0; obj_i < dpob_id.num_rows_held(); obj_i++)
      for (int arg_j = 0; arg_j < B.num_cols_held(); arg_j++)
         for (int j= 0; j < dpob_id.num_cols_held(); j++) {
            result += dpob_id[obj_i][j].value * B[j][arg_j].value;
            result[obj_i][arg_j] = result;
         }
   // return results to master.
   // two-copy new values.
}
```

Figure 4.12 Original, master and slave method definitions for `mat_mul()`.

methods of this class, and the generated code affected by the translations is shown in boldface.

The third translation, *result cardinality*, is more complex for this method class than the previous three method classes. In the simpler method classes, the result cardinality was dependent upon the `<subset_specifier>` of the object and the invocation of the method. In the case of local aggregate methods, the `<combination_rule>` annotation and the `<subset_specifier>` of the argument contribute information as well. The `<combination_rule>` annotation may have one of three values. If the `<combination_rule>` for the argument is **`1x1`**, then the compiler assumes that there are an equal number of subsets in the object and in the argument. Since they will be combined in a pairwise fashion,

number_of_results = subsets of object = subsets of argument.

However, if the combination rule is **`1xN`** or **`Nx1`**, then:

number of results = subsets of object * subsets of argument.

The *argument distribution* translation is analogous to the *object distribution* translation described for simple aggregate (**`ROW`** or **`COL`**) methods. However, it is applied to the data parallel argument of the method as opposed to the object upon which the method was invoked. Again, any communication generated by this translation occurs before the operation is invoked at the slaves (lines 17 and 18). However, since the data of an argument object is restricted to read-only access, the redistribution does not need to be undone as in the case of the simple aggregate (**`ROW`** or **`COL`**) methods.

The final translation required for the method complexity classes is *loop nesting*. This translation ensures that the iteration over both the invoked object and the argument object proceeds correctly. The `<combination_rule>` indicates the type of loop nesting required for the operation. For a **`1x1`** annotation, a single iteration control is required, and the same index is used for both the object and the argument. For a **`1xN`** annotation, two loops are needed. The outermost loop governs iteration over the object, and the innermost loop handles iteration over the argument. Consequently, the indexes used for the two objects correspond to the control loop index variables respectively. The translated body of the method should be at the innermost nesting level. This scenario is demonstrated in our example on lines 30-33. For an **`Nx1`** annotation, the nesting of the object and argument control loops is reversed.

Chapter 5 Conclusions and Future Work

In this thesis, our objective was to design a language in which both data and control parallelism are easily expressible and readily usable in conjunction with one another. The approach we used was to augment an existing control parallel language, the Mentat Programming Language, with data parallel extensions. Our extensions improve on previous work with data parallel languages in several ways. We generalize a number of known mechanisms and provide more flexibility in terms of data set manipulation. In particular, we generalize reduction operations, allow subsets of data parallel objects to be treated as data parallel objects themselves, allow flexible treatment of data set border regions by the programmer, and provide automatic generation of iteration control constructs. Our annotations allow elegant expression of data parallel method properties. Finally, we describe the requisite translations needed to map data parallel constructs to a control parallel paradigm.

Continuing work needs to be done in the areas of nested data parallelism, promotion of data parallel objects to first-class citizens, support of dynamic redistribution controlled by the programmer, and consideration of more complicated data set organizations. The compiler needs to be built both to test our translations and to expose issues that have escaped our notice. To truly prove the efficacy of a combined task and data parallel language, we would like to implement an application which requires both forms of parallelism.

Bibliography

[1] G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin-Cummings Publishing Co. Inc., Redwood City, CA,1994.

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

[4] A.A. Chien and W.J. Dally, “Concurrent Aggregates (CA),” *Proceedings of the Second ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, March, 1990, Seattle, Washington, pp. 187-196, .

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

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

[7] 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.

[8] A.S. Grimshaw, J.B. Weissman, and W.T. Strayer, ‘‘Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,’’ submitted to *ACM Transactions on Computer Systems*, Jul., 1993.

[9] A.S. Grimshaw, E.A. West, and W.R. Pearson, “No Pain and Gain! - Experiences with Mentat on Biological Application,” *Concurrency: Practice & Experience*, Vol. 5, No. 4, Jun., 1993, pp. 309-328 .

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

[11] P.J. Hatcher et al, “Data-Parallel Programming on MIMD Computers,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 377-383.

[12] W.D. Hillis, G.L. Steele, Jr., “Data Parallel Algorithms”, *Communications of the ACM*, Vol. 29, No. 12, 1986, pp. 1170-1183.

[13] J.F. Karpovichet al., “A Parallel Object-Oriented Framework for Stencil Algorithms,” *Proceedings of the Second Symposium on High-Performance Distributed Computing*, July, 1993, Spokane, WA.

[14] 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.

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

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

[17] 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.

[18] N. Nedeljkovic and M.J. Quinn, “Data-Parallel Programming on a Network of Heterogeneous Workstations,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, Sept., 1992, Syracuse, NY, pp. 28-36.

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

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

[21] 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.

[22] J.B. Weissman and A.S. Grimshaw, “Network Partitioning of Data Parallel Computations,” to appear in *Proceedings of the Symposium on High-Performance Distributed Computing (HPDC-3)*, August, 1994, San Francisco, CA.