Compiling Dataparallel C for Efficient Execution on Tightly Coupled Multiprocessors

Bradley K. Seevers
Major Professor: Dr. Michael J. Quinn

A Research Paper Submitted in
partial fulfillment of the requirements for the degree of
Master of Science

Department of Computer Science
Oregon State University
Corvallis, OR 97331

Presented November 10, 1991

Compiling Dataparallel C for Efficient Execution on Tightly Coupled Multiprocessors

Bradley K. Seevers
Major Professor: Dr. Michael J. Quinn

A Research Paper Submitted in
partial fulfillment of the requirements for the degree of
Master of Science

Department of Computer Science
Oregon State University
Corvallis, OR 97331

Presented November 10, 1991

Acknowledgments

Several people merit a special mention for their involvement in the development of the Dataparallel C compiler.

I would like to thank Dr. Michael Quinn for giving me the opportunity to design and write the Dataparallel C compiler, and for not showing any outward signs of doubt (even when the compiler implementation schedule slipped six months). Also for the occasional beer when something went exceptionally well. A thanks to my other committee members, Dr. Paul Cull and Dr. Tim Budd, for serving on my graduate committee, and for their suggestions pertaining to improving this manuscript. Thanks also to Ray Anderson and Guenter Mamier. Ray's heroic efforts in breaking my compiler, especially after each time I had fixed the last bug, were greatly appreciated. Guenter was cursed with the task of using my compiler while it was still in its infancy, and rather buggy.

I would like to extend a special thanks to my wife Sheila for kicking me when I started thinking out of control, and also for all those late night hot chocolates.

I also appreciate the generosity of the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory for allowing me access to their Sequent Symmetry, where most of the performance measurement were made.

This research was supported by National Science Foundation grant DCR-89006622.

Contents

Abstract

We describe a set of data flow techniques and code transformations that translate a single instruction stream, multiple data stream (SIMD) Dataparallel C program into a semantically equivalent single program, multiple data stream (SPMD) C program suitable for execution on shared memory multiprocessor computers, such as the Sequent Balance and Sequent Symmetry. The technique consists of identifying those areas in the original synchronous SIMD program where barrier synchronizations must be enforced to preserve the semantics, and then rewriting the program as a loosely synchronous SPMD C program that includes calls to barrier synchronization library routines. The run-time model used by the translated program is also presented. We discuss a Dataparallel C compiler we have implemented using our proposed methodology. Finally, we present some performance results for our compiler, and we discuss techniques to improve these results.

1 Introduction

We are convinced that much of the difficulty in programming parallel computers is a direct result of the paradigms being used. We now describe a paradigm that we feel solves many of the problems of parallel programming. With the development of massively parallel SIMD machines, such as the Connection Machine, a new style of parallel programming has emerged: *data-parallel programming*. We characterize data-parallel programming as having the following features:

Explicitly Parallel Data. The programmer declares parallel data that is to be distributed among the processors. The programmer may also specify how the data is to be distributed.

Explicitly Parallel Code. The programmer explicitly writes parallel and sequential code. Parallel code can be any general code, and is not limited to expressions. The compiler performs no automatic parallelization of the source code.

Virtual Processors. Associated with each parallel data item is a virtual processor. Virtual processors are necessary, because the number of parallel data items may exceed the number of physical processors. There is no limit to the number of virtual processors available. If there are more virtual processors than physical processors, they will be emulated by the translated program and not the operating system.

SIMD Semantics. Parallelism is achieved by virtual processors applying identical operations to their associated data items in parallel. Programs execute as if there is a single instruction counter for all virtual processors. Semantically, all virtual processors work lock step. In this way operations can be applied across collections of data in parallel.

Shared Memory Model with Global Name Space. No explicit communications are necessary in a data-parallel program. Instead, virtual processors communicate through general pointer or array references. In parallel code it is possible to access or assign sequential data items declared in outer scopes, just as in sequential programs. In sequential code it is possible to access or assign the values of parallel data items. Virtual processors can also access and assign each other's parallel data.

Virtual Processor-Oriented Code. Code is written at the virtual processor level, as if the programmer were writing the code for a single virtual processor. The code is then executed by all virtual processors in parallel.

Deterministic Execution. This is a consequence of the SIMD semantics and greatly reduces programming errors and simplifies the debugging process.

We feel that the data-parallel style of programming solves many of the problems of the other paradigms. The shared memory model removes the potential for communication bugs. SIMD semantics removes race conditions and guarantees determinism. The amount of data parallelism scales with the problem size, making it easy to utilize large numbers of processors. Programming from the local view, we believe, is easier for the programmer than programming in the global view, which is the paradigm for parallel array based languages, such as Vector C [15], Parallel Pascal [26], and parallel implementations of APL [2], because the programmer has more direct control. Also, we feel that the features of data parallel programming languages make the programs easier to debug and maintain. In a study conducted by Fox [6], he found as many as 83% of existing scientific applications are amenable to a data-parallel solution. It seems that a data-parallel solution to a data-parallel problem is a natural solution. We are *not* presenting data-parallel languages as *the only solution* to parallel programming; rather, our claim is that in many cases data-parallel programming provides the programmer with a manageable solution. For more information on data-parallel programming and data-parallel algorithms see [12, 13, 21].

Our research has focused on the compilation of data-parallel languages for efficient execution on multiple instruction stream, multiple data stream (MIMD) architectures. More specifically, we have developed a Dataparallel C compiler for the tightly coupled Sequent Symmetry and Balance multiprocessors. Our compiler uses data flow techniques to build *use-def*, *def-use*, and *def-def* dependency chains for all expressions in parallel code. These data dependency chains are then used to locate all places in the original code where barrier synchronizations need to be enforced to preserve the semantics of the original Dataparallel C program. Once these synchronization points are known, our compiler uses a sequence of code transformations to rewrite the original Dataparallel C code as a semantically equivalent, loosely synchronous SPMD C program that includes calls to the Sequent parallel programming library for parallel data and process management. Additionally, the compiler must translate all Dataparallel C programming constructs not found in C into semantically equivalent C code.

At first thought, the idea of compiling a SIMD language for a MIMD machine may seem odd, but we believe that for many applications SIMD languages on MIMD machines have advantages over SIMD languages on SIMD machines and MIMD languages on MIMD machines. These advantages are:

SIMD languages have many desirable characteristics not found in other parallel programming paradigms, as discussed earlier.

They provide a convenient means for writing data-parallel programs for MIMD machines, which traditionally have had only low-level support for parallel programming.

Communications and competing memory references may be spread out temporally on a MIMD machine due to their asynchronous behavior, thus reducing contention for resources and improving efficiency

It is often possible to more effectively utilize the CPUs of a MIMD machine in conditional code than those of a SIMD machine by relaxing the SIMD constraints, if the semantics can be preserved.

This project is not the first attempt at compiling SIMD languages for MIMD architectures. Quinn and Hatcher describe a C* compiler for the nCUBE muilticomputer in [7, 20, 21, 22, 23]. In [8, 10, 11, 9, 21] Quinn and Hatcher present performance figures for a Dataparallel C compiler on the nCUBE and Intel iPSC/2 multicomputers. Similar work is also being done at NASA ICASE on the data-parallel language Kali, targeted for multiprocessors [14].

We use a variant of the *SelectSyncs* algorithm developed by Quinn and Hatcher [24] to determine a minimal set of barrier synchronizations necessary in the translated Dataparallel C program. We have modified the algorithm to allow synchronization points to be chosen more optimally.

Array processing languages, such as Vector C, Parallel Pascal, and APL have been developed and parallelized. While these languages do share many similar concepts with Dataparallel C, such as explicit parallelism and virtual processors, we feel these languages are not true data-parallel languages, because solutions must be coded with the global view, and not the local view.

Several true data-parallel programming languages have been developed for SIMD machines, such as *Lisp [29] and C* [27], both by Thinking Machines Corp. The development of C* was significant, because it was an upwardly compatible data-parallel extension to the highly popular and efficient programming language C. C* extends C in relatively few ways. It provides mechanisms to declare both parallel and scalar data items, and to write both parallel and scalar code. Scalar code behaves exactly as standard C code, and parallel code behaves in a natural data-parallel extension to the scalar model. It provides the programer a virtual machine model with a scalar front-end, where sequential code and data reside, and a back-end array processor, where parallel code and data reside. The back-end array processor always has the required number of virtual processing elements. Dataparallel C is a variant of C*. For a more complete description of the C* programming language see [27].

The remainder of this paper is organized as follows. In section 2 we provide a brief introduction to Dataparallel C. Section 3 gives a conceptual overview of the compilation process. Here we describe the run-time model, and give a brief description of how the compiler translates Dataparallel C to C. In section 4 we discuss the actions of our compiler during the scan and parse phase. Section 5 describes the data flow process and gives some examples of the data dependencies that must be preserved. Section 6 shows how the data flow information is used to select a minimal set of synchronization points needed to preserve the SIMD semantics of the Dataparallel C program. In section 7 we present the code transformations and several important optimizations implemented by our compiler. Section 8 describes how peephole optimization cleans up the code generated by the code transformations. Section 9 describes the unparse phase, where many of the constructs unique to Dataparallel C are translated into C. In section 10 we evaluate the performance of our compiler, both in the code it generates, and the time it requires to compile Dataparallel C codes. Section 11 outlines future work that could be done on both the language Dataparallel C and our compiler. Some of the shortcomings of Dataparallel C and optimizations that could improve the performance of the translated programs are discussed. We end in section 12 with a summary of our work, and draw some conclusions from our results.

2 The Dataparallel C Programming Language

Since Thinking Machines first introduced the C* language, they have developed the C* version 6.0 language [28], which differs substantially from the original. We have made several small changes to the original C* language, and renamed it Dataparallel C to avoid confusion with the new C* language. We now provide a brief language description for Dataparallel C. For an in depth description see [21].

Dataparallel C is an extension of ANSI C. Any ANSI C program should compile and behave as expected. To add parallelism, Dataparallel C provides three areas of extension to ANSI C.

1. Programmers can declare parallel data types and instances of these parallel data types (parallel data). The programmer can also specify how this data is to be distributed.
2. Programmers can write parallel code and parallel functions.
3. Several new expression operators have been added to the language.

Parallel data items can be declared in two ways. The first is with the *domain declaration*. A domain declaration is similar to a structure declaration, and can be used to declare both instances of parallel data and parallel data types. Each instance of a parallel data type is associated with a virtual processor. For example, the code below declares a domain type `foo` and 10000 instances of this type in the array `poly_data`. Each data instance has an associated virtual processor, an integer named `bar`, and an array of 20 floats named `baz`.

```
domain foo {
  int bar;
  float baz[20];
} poly_data[100][100];
```

The second mechanism by which parallel data can be declared is by declaring data items locally inside parallel code, in the normal C blocked scoping fashion. Variables declared in this way are created with one instance for each virtual processor (each virtual processor has its own variable).

Virtual processors are distributed over the physical processors using one of seven mappings; `contiguous`, `contiguous_row`, `contiguous_col`, `interleaved`, `interleaved_row`, `interleaved_col`, and `blocked`. The programmer can optionally specify a domain distribution when the domain type is declared, as in:

```
domain contiguous foo {
  int bar;
  float baz[20];
} poly_data[100][100];
```

If no distribution is specified, our compiler defaults to `interleaved`, causing each physical processor to emulate the *n*'th virtual processor, where *n* is the number of physical processors. Contiguous mappings cause the virtual processors for chunks of neighboring data items to be emulated on the same physical processor. Our compiler implements only two different distributions; the contiguous and blocked mappings are grouped together, and the interleaved mappings are grouped together. This is because of how we actually emulate virtual processors, which is discussed later. At compile time it is possible to change the default mapping from `interleaved` to `contiguous`, using a command line switch.

There are also two methods of specifying parallel code. The first is the *domain select* statement. The syntax of the domain select is

```
[domain domainname].statement
```

where statement can be (and usually is) a compound statement that contains the code to be executed in parallel. The reference to `domainname` must be a pre-declared domain type tag name, and not an instance name. Code outside a domain select behaves as normal sequential code, but when execution enters a domain select, all the virtual processors of type `domainname` become active and execute the domain select statements and expressions together in a SIMD fashion. All legal C statements and expressions are permitted in a domain select, except the goto statement. In [20] Quinn and Hatcher describe why the goto cannot be implemented efficiently in SIMD languages compiled for MIMD architectures. For this reason, goto is not allowed in parallel code. Due to the infrequent use of goto, we hardly feel this is a serious restriction.

Domain selects cannot be nested, either statically or dynamically through function calls. However, it is legal to invoke parallel functions, discussed next, of the same domain type from inside parallel code.

The second method of declaring parallel code is through parallel, or *member function* declarations. A member function declaration is similar to its C++ counterpart. A function prototype for all member functions must appear in the domain declaration. When the actual function is declared, its name must be prefixed with the domain type, such as

```
domain foo {
  int i;
  float bar[20];
  int member_func();
} baz[100][100];

int foo::member_func()
{
  ...
}
```

Again, all legal C statements and expressions are allowed in member functions except goto. Additionally, member functions may be invoked from either sequential or parallel code. Domain selects are not permitted in member functions, as this would nest parallelism.

When a control structure, such as an if or while statement, is executed in parallel code, only those virtual processors for which the condition is true execute the body of the structure, while the others "turn off" and wait for the active virtual processors to finish. In the case of an if with an else clause, the virtual processors for which the condition was false wait at the else clause until all those virtual processors for which the test was true have finished the then portion. Those that were active now become inactive and wait at the bottom of the else portion, while those that were inactive now become active and execute the else.

To invoke member functions from parallel code, either the form

```
domain_type_tag::member_func_name(args)
```

or

```
member_func_name(args)
```

can be used. Since parallelism cannot be nested, domain_type_tag must be the same as the current domain select, and hence, it need not be specified. When calling a member function from sequential code, the

```
domain_type_tag::member_func_name(args)
```

form must be used. If a member function is called from parallel code, only those virtual processors that are currently active execute the function. If a member function is called from sequential code, all virtual processors will become active. The return value of the lowest numbered virtual processor (the one for the first instance of the domain) will be used as the sequential value of the member function.

Dataparallel C introduces some new expression types and operators to the C language. The first of these is the keyword `this`. Similar to C++, `this` is defined to be a pointer to each virtual processor's local data. For example, in parallel code

```
this->bar[2]++;
```

would cause every virtual processor to increment the value of the third element in its own `bar` array. Virtual processors can access data values of other virtual processors through pointer arithmetic on `this`. The expression

```
this->i = (this+1)->i;
```

would cause all virtual processors to read their right-hand neighbors value of `i`, and then assign it to their value of `i`. The SIMD semantics of the language guarantee that all virtual processors will have read their neighbor's value of `i` before any assign the value to their local copy of `i`. The keyword `this` is only allowed in parallel code.

Often, `this` can be omitted. The name resolution rules in parallel code specify that member names of the active domain type are searched first, and then names of outwardly scoped variables are searched. The above example could have been written as

```
i = (this+1)->i;
```

again, similar to C++. Dataparallel C provides no mechanism to access outwardly scoped variables in the case of name aliasing.

To further aid the programmer in accessing values of neighboring virtual processors, the language provides some pre-defined *communication macros*. These are `successor()`, `predecessor()`, `north()`, `south()`, `west()`, `east()`, `northwest()`, `northeast()`, `southwest()`, `southeast()`, `up()`, and `down()`. All communication macros are only valid if they make sense for the shape of the domain. For single-dimensional arrays, only `successor()` and `predecessor()` are valid. For two-dimensional arrays, they all are valid up through `southeast()`. For three-dimensional arrays and higher, they are all valid. These macros take no arguments, and evaluate to a pointer to the desired virtual processors data items of the same type as the current domain select. These macros also provide toroidal wraparound for boundary virtual processors. Communication macros are only valid in parallel code.

We often refer to parallel variables as *poly* variables and sequential variables as *mono* variables. Two corresponding type qualifiers have been added to the language, `poly` and mono.

It is possible to use the `mono` type qualifier to declare mono data in parallel code, which has the same effect as declaring the variable in sequential code. It is also possible to use `mono` and `poly` to cast the type of an expression to either mono or poly. The `mono` and `poly` qualifiers are only legal in parallel code.

For convenience, Dataparallel C adds two new operators to C, *min* and *max*. They both have a binary and an assignment form. Min is <? or <?=, and max is >? or >?=. These are similar to the common min and max macros, with the exception that they only evaluate their arguments once. The min and max operators are allowed in either sequential or parallel code.

It is often useful in parallel code to be able to perform some kind of a *reduction* operation across parallel data, such as finding the sum, or max of a domain member. To provide this functionality, Dataparallel C has overloaded most of the assignment operators. The reduction operators are:

+=	Sum reduction
−=	Negative of the sum reduction
*=	Product reduction
/=	Reciprocal of the Product reduction
|=	Bitwise OR reduction
&=	Bitwise AND reduction
^=	Bitwise XOR reduction
<?=	Min reduction
>?=	Max reduction

Each reduction operator has both a unary and an assignment form. The expression

```
i = <?= b[0];
```

contains a unary min reduction. In evaluating the unary min operator, all currently active virtual processors participate in finding the minimum of the `b[0]` values, they then assign this value to their own `i`.

The rules for the assignment reduction operators are not so simple. For example

```
i <?= b[0];
```

is not a reduction at all, but instead a minimum assignment. The reason for this is that `i` is a poly variable, so each virtual processors would simple take the min of its `i` and `b[0]` values, and assign the result to its local `i`. However, if `i` had been a mono variable, a reduction would have been performed.

To better understand when an assignment is a reduction, we introduce the "as if serial" rule. This rule specifies the semantics of assignment in parallel code are as if the virtual processors executed it in some serial order, with no two executing at the same time. The arguments to the assignment operator are still evaluated with SIMD semantics. This may seem to break from the SIMD semantics of Dataparallel C, but it is necessary to fully understand what function any particular assignment expression performs. Assignment operators only perform reductions if two or more virtual processors assign to the same *lval*. When the number of lvals is exactly one, a reduction is performed. If the number of lvals is equal to the number of active virtual processors, a regular assignment type operation is performed. It is possible for the number of lvals to be less than the number of active virtual processors, but greater than one. We call this a *multi-reduce*, because the are multiple sub-reductions taking place. The assignment reduction form also differs from the unary form in that the *rval* of the left argument also participates in the reduction. In actuality, we can implement most assignment operators in parallel if at compiler time we can determine the uniqueness of the lval being assigned, despite the as if serial rule. Reduction operators are only allowed in parallel code. Reduction assignments are not truly deterministic, because the ordering of the as if serial rule is not specified. Potentially different round off and overflow errors could result from floating point reductions performed in different orders.

Constant reductions may also be performed, and may be useful for such things as determining the number of active virtual processors, as in

```
num_active = += 1;
```

Normally constants are mono expressions, but in the presence of a unary reduction operator they are automatically coerced to a poly.

We have added array assignment to Dataparallel C. If the left hand side of an assignment operator evaluates to an array of known dimensions, and the right hand side evaluates to an array or pointer, the entire left hand side is assigned from values pointed to by the right hand side. Array assignments are permitted in parallel or sequential code. The reason for adding array assignment to the language was to maintain compatibility with a Dataparallel C compiler being developed at the University of New Hampshire, and is mostly a convenience.

Below we present a complete example of a Dataparallel C program to compute an estimation of π using the rectangular rule to integrate $4/(1+x^2)$ between 0 and 1.

```
#define INTERVALS 400000

domain span { double x; } chunk[INTERVALS];

main ()
{
   double sum;                          /* Sum of areas       */

   [domain span].{
      double width = 1.0/INTERVALS;     /* Width of interval */

      x = (this - &chunk[0] + 0.5)*width;
      sum = += (4.0/(1.0 + x*x));
   }
   sum *= 1.0/INTERVALS;
   printf ("Estimation of pi is %2.12f\n", sum);
}
```

This program declares a domain of type span with one member x of type double. It then declares a one dimensional array with 400000 elements named chunk of type span. This establishes 400000 virtual processors organized as a *ring*, each with a double precision floating point variable named x. These x's are poly data, but chunk is a mono variable (there is only one instance of chunk).

In the function main, sum is a normal C mono variable (only a single instance). The domain select [domain span] causes all 400000 virtual processors to start executing synchronously. The declaration

```
double width = 1.0/INTERVALS;
```

declares a local double precision poly variable (every virtual processor has its own variable), and initializes it to 1.0/4000000. In the next statement

```
this - &chunk[0]
```

evaluates to a unique integer for each virtual processors that corresponds to its position in the chunk array. Each virtual processor uses this value to compute its x position in the integral, and assigns the result to its x member variable. In the next line, a sum reduction is performed to add up the areas for each virtual processors rectangle. The value is then assigned to sum, and the domain select is exited, returning to sequential code. The remainder of the program is standard C.

Note the single locus of control, much like a sequential language. We feel this makes programs easier to write, debug, and understand.

3 Conceptual Overview of the Compilation Process

Given the problem of executing a SIMD program on a MIMD machine, there are several possible solutions. One would be to translate the parallel sections of code into an intermediate form which is then interpreted by a SIMD emulator running on the target machine. The problem with this approach is that interpreting the intermediate code would be much too slow, negating the performance gain of parallelism.

Another similar approach would be to translate the data-parallel code into a series of calls to SIMD type library support routines. While this would undoubtedly be faster than the first approach by reducing the run-time overhead of interpreting the intermediate code, it would still have performance problems due to the high overhead of the number of calls required and the inability to optimize the called routines to the specific needs of the executing code.

A third approach would be to translate the Dataparallel C program into another high level parallel programming language already running on the Sequent multiprocessors, such as C Linda. The master, or front-end process, could eval off a stream of work to be done by the subtasks in parallel. Again, the problem would be performance. While portability would be high, the overhead of this execution model would probably outweigh its benefits.

The approach we have taken is to translate the SIMD Dataparallel C code into a semantically equivalent SPMD C program containing calls to the Sequent parallel programming library for parallel process and data management. We feel this overcomes the previous problems by allowing extremely lightweight virtual processor emulation (a simple `for` loop), the resulting code is a direct mapping of the original Dataparallel C code, and it allows a high degree of optimization. We do as much as possible at compile time to reduce the amount of work done at run-time. The translated code is closer to the actual underlying architecture than the other methods, resulting in higher performance.

The Sequent C compiler and parallel programming library provides some low level mechanisms for parallel process and data management. The C compiler has been extended to allow the declaration of global shared memory variables. Such variables will be readable and writable by all parallel processes. To declare a shared variable, the type storage class specifier `shared` must be used, such as in

```
shared int i, j, k;
```

The shared storage class specifier is only allowed for global variables.

The library routines provide mechanisms for controlling access to shared variables, creating and killing parallel processes, enforcing barrier synchronizations, and other miscellaneous tasks. The routines we use are explained below:

`m_set_procs(nprocs)`	Sets the number of child processes
`m_fork(func[,arg,...])`	Execute subprocess in parallel
`m_kill_procs()`	Kill all child processes
`m_single()`	Suspend child processes
`m_multi()`	Resume child processes
`m_sync()`	Perform a barrier synchronization
`m_lock()`	Locks the bus
`m_unlock()`	Unlocks the bus
`m_get_myid()`	Returns a unique id for each physical processor in the range 0 – nprocs-1

For more information on the Sequent parallel programming model, see [18].

3.1 The Compiler's Strategy

Enforcing strict SIMD semantics on a shared memory multiprocessor MIMD machine is expensive. The program must use barrier synchronizations to guarantee all of the asynchronous physical processors execute synchronously. Since data-parallel languages are not only SIMD on the statement level, but also the expression and subexpression level, enforcing this level of synchronous execution would require a significant number of barrier synchronizations, nullifying any potential speedup that could be gained by parallel execution.

The goal of our compiler is to relax the SIMD execution of the translated program while preserving the SIMD semantics of the original Dataparallel C program. This can be done because barrier synchronizations only need to be enforced when there are interactions between virtual processors. Virtual processors interact through expressions, so virtual processors interact through modifying and reading common data items, both local and global. Where there are no interactions, no barrier synchronizations are necessary and processors can execute at full speed. To detect these potential interactions, the compiler performs extensive data flow analysis. The following example shows a simple example of how the SIMD model of execution can be relaxed for a significant performance gain, while preserving the semantics of the original Dataparallel C statement.

Let us examine the case where the domain is a two dimensional grid with toroidal wrap around. If every cell was to set its value to the average of its four neighbor values, we could use the following assignment:

```
i = (north()->i + south()->i + east()->i + west()->i)/4;
```

Semantically, these steps are taken to evaluate the statement:

1. All virtual processors evaluate `north()`, yielding a unique value for each.
   barrier synchronization
2. All virtual processors evaluate `north()->i` as an lval (to get the address of `north()->i`), again getting a unique value.
   barrier synchronization
3. All virtual processors read `north()->i`, using the address obtained in 2.
   barrier synchronization
4. All virtual processors repeat steps 1 through 3 for south.
5. All virtual processors add the values of `north()->i` and `south()->i`.
   barrier synchronization
6. This is repeated for east and west
7. All virtual processors divide this sum by 4.
   barrier synchronization
8. All virtual processors evaluate `i` as an lval.
   barrier synchronization
9. All virtual processors store the average of their neighbors values in their `i` value.

This yields the correct results, because all virtual processors execute in a SIMD fashion. They have all read their neighbors values of `i` before any values of `i` are modified. However, it is easy to see that the following evaluation scheme will yield the same correct results:

1. `poly_tmp = (north()->i + south()->i + east()->i + west()->i)/4;`
   barrier synchronization
2. `i = poly_tmp;`

This works, because all virtual processors have again read their neighbors values of `i` before any values of `i` are modified. We have maintained the SIMD semantics while reducing the number of barrier synchronizations from 17 to one. This is a substantial savings, even in this single small code example.

Once all the dependencies for a program are known, it is possible to divide the domain selects up into multiple domain selects, such that all data dependencies span at least one domain select exit and entry. If this is done, all data dependencies are inter-domain dependencies, and there are no intra-domain dependencies, so barrier synchronizations only need to be enforced between domain selects, requiring only the physical processors to synchronize and not the virtual processors. This fits the physical machine model more closely, allows the virtual processors to be more easily emulated, and is in fact the approach we have taken. Dividing the domain selects is not always a trivial task, because it may involve splitting control structures and expressions, since a data dependency may be wholly contained in a single expression, `if` statement, loop structure, or `switch` statement. The methods used to divide domain selects are described later in section 7.

After performing the necessary domain subdivisions, since there are no intra-virtual processor intra-domain data dependencies, no barrier synchronizations need to be enforced in the domain selects, and the domain selects can be emulated with simple `for` loops. Since there are no dependencies between virtual processors in any domain select, virtual processors can be emulated in any order. Using a `for` loop instead of actual processes provides an extremely light weight virtual processor context switch.

The expression types unique to Dataparallel C (array assignment, reductions, min, and max) must also be translated into either regular C code or function calls. This is explained in more detail in section 9.

Since Dataparallel C has a global name space, it is also necessary to place all variables accessed in parallel code into shared memory, so that all physical processors will have access to the same data. Data flow is used to determine which variables other than domain instances must be placed in shared memory. The domain type declaration is translated into a structure declaration, and instances of this type are automatically placed in shared memory by the compiler.

3.2 The Run-time Model

The run-time model we have chosen is that of a single processor that executes the front-end code, and a group of processors that together with the front-end processor execute the back-end parallel code. When a Dataparallel C program begins execution, the front-end processor fires up some user specified number of processors (these are processors, and not processes), but they immediately put themselves to sleep. In this way they are ready, waiting for execution to enter parallel code. When it does, the front-end processor wakes them up, and all processors, including the front-end processor, emulate their share of the virtual processors in parallel. Since the domain selects have been divided such that there are no dependencies in any single domain select, the processors can emulate virtual processors at full speed. There are several possibilities when execution exits a domain select. If it is the end of the original domain select, all processors except the front-end processor go back to sleep (this also performs a barrier synchronization). If execution is between two split domain selects, then all processors participate in a barrier synchronization, remain active, and then enter the next domain select. The process repeats throughout the execution of the Dataparallel C program. In this way virtual processors are emulated in parallel, and the semantics of the original program are preserved.

3.3 The Structure of the Compiler

Our Dataparallel C compiler breaks the process of translating a Dataparallel C program into a SPMD C program into 9 major phases:

1. Scan and parse phase.
2. Data flow phase.
3. Find a minimal set of barrier synchronizations necessary to preserve the semantics of the original program – an optimization.
4. Scalarization – an optimization. Renaming and moving variables.
5. Domain invariant code motion – an optimization.
6. Fix up break/continue/return in parallel code – a code transformation.
7. Insert barrier synchronizations and perform code transformations.
8. Peephole optimizations.
9. Emit C code.

4 Scan and Parse Phase

The first phase of our compiler performs scanning and parsing to build a conventional abstract syntax tree. Also performed in this phase are complete type checking, constant folding, and expanding the communication macros. During type checking, the types of all expression nodes are computed and cached in fields of their respective expression nodes for later use. For example, if a is of type `int`, given the expression tree for `a + 3.2` the expression node for a would have type `int`, the node for `3.2` type `double`, and the + node would also have type `double`. If any type or expression errors are detected during this phase they are reported. Also cached in each identifier reference node is its symbol table information. The purpose of caching these attributes is to speed up later phases that need to be able to determine the types of expression nodes, and get at the symbol table entries for all identifier references. Caching eliminates the need to recompute this information, saving time. Additionally, all identifier references that are really member references are rewritten as member references. For example, if `i` is a domain member, the expression `i =` *exp* would be rewritten as `this->i =` *exp*. If `i` was instead declared locally inside a domain select or member function, it would not be rewritten, in fact `this->i` would be an illegal reference, because `i` is not a member!

For each type of communication macro used (i.e. `north()`, `south()`, etc.), a special member is added to the current domain type. The type of these special members is defined to be a pointer to an instance of the domain type. The communication macros are expanded into pointer references using these special pointer members. The compiler keeps track of which communication macros were used for each domain type, and code will be emitted during the unparse phase to initialize them. This scheme requires that the addresses only need to be computed once, and gives a very quick mechanism for each virtual processor to access its neighbors. To illustrate, the macro reference `north()->i` expands into something like `this->__north->i` which is much more efficient than expanding it into an expression that recomputes what the address of the north neighbor would be each time it is used. Computing addresses of neighbors is expensive, because the toroidal boarder connections must be supported.

When the parse tree for a member function is built, we make several important changes to the member function. The body of the member function is wrapped in a domain select, providing virtual processor emulation in the member function itself, and not where it is invoked from. If the member function is invoked from parallel code, not all the virtual processors may be active. To account for this, we wrap the body of the domain select with a `if` statement that uses a new poly temporary variable we add to the domain declaration, as its test condition. The poly test variable will be initialized prior to each invocation of the member function. As an example, consider:

Dataparallel C Source:

```
foo::bar()
{
  statement_block;
}
```

Becomes:

```
foo::bar()
{
  [domain foo].{
    if (poly_tmp_active) {
      statement_block;
    }
  }
}
```

The reasons for modifying member functions in this way is to provide a common form for parallel code; it will always be in a domain select. This allows barrier synchronizations to be enforced in the member functions much more easily, and the same transformations can be used for both member functions and domain selects. How member functions are invoked is discussed in section 7.

5 Data Flow Phase

To preserve the semantics of the translated code, all data dependencies between *different* virtual processors must be preserved, because we must guarantee that all virtual processors have finished using a data item before any modify it, and also that a variable is through being modified before any virtual processors use it. This is accomplished by first locating all inter-virtual processor data dependencies, then enforcing a barrier synchronization anywhere between the endpoints for each of the dependencies to ensure the execution order of the dependency endpoints. We are only interested in data dependencies between different virtual processors, because data dependencies for the same virtual processor are guaranteed to be executed in the proper order, but since different virtual processors may be emulated on different physical processors, there is no guarantee without barrier synchronizations for inter-virtual processor data dependencies. The data dependencies that must be preserved are *def-use*, *use-def*, and *def-def*. We now define these dependencies.

> **DEF-USE:** If the lval of variable a is required by virtual processor i at expression x, and the rval of variable a is required by virtual processor j, $i \neq j$, at expression y, and the definition of a at statement x is in the set of reachable definitions at statement y, this is a *def-use* dependency for variable a. This implies there must be a possible execution path from expression x to y that does not assign to a.

Def-use dependencies are important, because if a variable (either global or local) is modified by a virtual processor and later used by a different virtual processor, we must ensure that the variable is through being modified before it is used. By inserting a barrier synchronization between when it is defined and used, we guarantee that all virtual processors are through modifying it before any use its value.

Example of Def-Use Dependencies:

```
if (exp1)
  north()->poly = exp2;          Def of poly
mono = exp3;                     Def of mono
  ...
var1 = poly;                     Use of poly
var2 = mono;                     Use of mono
```

In this example there are two def-use dependencies, one between the poly variable in the expression `north()->poly = `exp_1 and the expression `var`$_1$` = poly` and another between the mono variables in the expression `mono = `exp_2 and expression `var`$_2$` = mono`. The `if` statement has no effect on the def-use dependencies, other than it creates multiple possible execution paths.

USE-DEF: If the rval of variable *a* is required by virtual processor *i* at expression *x*, and the lval of variable *a* is required at expression *y* by virtual processor *j*, $i \neq j$, such that the set of reaching definitions at *y* is a super set of the reaching definitions at *x*, this is a *use-def* dependency for variable *a*. This definition implies there must be an execution path from expression *x* to expression *y* that does not assign to *a*.

Use-def dependencies are important for a similar reason that def-use dependencies are important. Because of its similarity, we only give a simple example of a use-def dependency.

Example of Use-Def Dependency:

```
poly = east()->poly;
```

In this example, all values of poly are read by their west neighbors, and then all values of poly are assigned their neighbors value.

DEF-DEF: If the lval of variable *a* is required by virtual processor *i* in basic block *x* and also required by virtual processor *j*, $i \neq j$, in basic block *y*, $x \neq y$, and the definition of *a* in basic block *x* is in the set of reachable definitions for *a* in basic block *y*, this is a *def-def* dependency for variable *a*. This implies there exists an execution path from block *x* to *j* that does not assign to *a*.

We are concerned only with def-def dependencies between basic blocks, because for a def-def dependency within a single basic block, the first def would be a dead assignment unless there is a subsequent use before the second def, in which case there would be a def-use dependency that would ensure correctness. This is not true between basic blocks. We present an example to demonstrate why def-def dependencies are important.

Example of Def-Def Dependencies:

```
if (poly_exp1) {
  mono = exp2;
  poly = exp3;
} else {
  mono = exp4;
  east()->poly = exp5;
}
```

This examples shows how def-def dependencies can be created for both mono and poly data types. If there is not a barrier synchronization enforced between the `mono = `exp_2 and the mono = exp_4 statement, it would be possible for all mono = exp_4 statements to be completed before the last mono = exp_2 statement. The SIMD semantics of the language specify that all mono = exp_2 statements must be completed before any of the mono = exp_4 statements. To guarantee correctness, a barrier synchronization can be inserted anywhere between these two statements. Similarly, there is a def-def dependency between the poly = exp_3 and east()->poly = exp_5 statements, because the same data can be referenced by different virtual processors. Again, a barrier synchronization must be inserted between these two statements.

It should be pointed out that in the case of overlapping dependencies, a single barrier synchronization placed at any point in the intersection of the dependency chains is sufficient to preserve the semantics of all dependencies. From this we see that each of the previous examples only needed a single barrier synchronization.

These have all been very simple data dependencies. It is possible to make these dependencies arbitrarily complex through pointer arithmetic, array subscripting expressions, assignments, and control structures. It is easy for a programmer to make errors when faced with a few simple data dependencies; as they become increasingly complex and the program size grows, the importance of having the compiler perform this function becomes invaluable.

To correctly compile Dataparallel C we must compute use-def, def-use, and def-def dependencies for all parallel code. These dependencies must be correctly preserved in the translated program. Data flow information for the entire program is gathered immediately after the abstract syntax tree is built for the entire Dataparallel C source code.

Use-def, def-use, and def-def dependency chains are constructed from the parse tree during data flow using a syntax-directed solution to data flow, similar to that described [1]. This information is then stored in the parse tree. The syntax-directed data flow method can be used because Dataparallel C flow graphs are guaranteed to be reducible due to prohibiting the use of the goto statement in parallel code. Data flow information is only gathered for expressions in parallel code. It would be possible to perform inter-procedural data flow for functions invoked from within parallel code, but this has not been implemented. The data flow routines analyze all expression tree nodes to determine if a node is a use, a def, or neither, of any data item. Any expression that is an rval is a use, and any expression that is an lval is considered a def. Some expressions are considered both a use and a def, such as i in the expressions i++, i += 2, and func(&i). We consider the last example both a use and a def of i, because without doing intra-procedural data flow analysis we have no means of determining the correct use of i. In the case of array and structure/union/domain member references, these are considered uses or defs of the base object, and not the specific member or instance. For example, we consider foo->bar to be a reference to foo, and not to bar. This level of resolution is handled later.

Four distinct types of data flow chains are built. Use-def and def-use chains are built where the same data item may be accessed by different virtual processors at different times. Def-def chains are built for data references to the same data item by different virtual processors in different basic blocks. The reasons for these data flow chains has been previously discussed. We build a fourth type which is def-use chains where the same data is accessed by the same virtual processor. The reason for collecting this information will be discussed under the scalarization optimization in section 7.1, and is not essential for the correct compilation of Dataparallel C.

When building the chains, we need to be able to determine if any two expression nodes are referencing the same data, and if so if the references are by the same virtual processor, or different virtual processors. Two oracle routines have this responsibility; *is_same_data* and *is_same_vps*.

The function is_same_data takes two pointers into the parse tree and attempts to determine if they are referencing the same instance of a data item. This is not always an easy task. In the case that is_same_data is unable to determine the exact data items being referenced (references through pointers for example), it always assumes they are the same, which guarantees the semantics will be preserved, but may not be optimal. It is the responsibility of is_same_data for determining if two array references are the same, and if member references are referencing the same member. Given the domain declaration

```
domain foo {
  domain foo *ptr;
  int i, j[20];
} bar[100], *baz;
```

The following are examples of the same data references:

References to `i`:	References to `j`:	References to `i` and `j`:
`i`	`j`	`*this`
`this->i`	`this->j[2]`	`*ptr`
`ptr->i`	`ptr->j[2]`	`*baz`
`base[exp].i`	`base[exp].j[2]`	`base[exp]`
`(this+exp)->i`	`(this+exp)->j`	
`this->ptr[exp].i`	`this->ptr[exp].j[2]`	
`baz->i`	`*baz->j`	

Function is_same_data tries to determine if the bases of the two expressions are the same, and if they are, do they reference the same member items. The routine is_same data must not only try to determine if parallel data references are the same, but also mono variable references.

The oracle is_same_vps is a similar routine, taking two pointers into the parse tree and attempting to determine if they are expressions that can be referenced by different virtual processors, such as `this->i` and `(this+1)->i`, or the same virtual processor. Again, this can be very difficult, and in the case that is_same_vps is unable to conclude if the virtual processors are the same for the two expressions, it always assumes they are not and returns false.

The key to is_same_vps is another routine, *is_refable_by_this*. The helper function is_refable_by_this takes a pointer to an expression in the parse tree and tries to determine if the expression is referencing something that could be referenced off of `this`. It returns either false, or true and whether or not it is a constant offset from `this`. If it is a constant offset from `this` it also returns that offset. This information is used by is_same_vps to determine if two expressions are being referenced by the same virtual processor. If two expressions are referencable by `this`, and they have identical offsets, then is_same_vps returns true. Otherwise is_same_vps returns false. Some examples of identical virtual processor references are:

The Reference:	Is the same virtual processor as:
`this->i`	`i`
`(this+2)->i`	`(this+2)->j`
`base[this-&base+3].i`	`*(this+3)`
`i`	`j`
`j[4]`	`*ptr`
`successor()->i`	`(this+1)->i`
`ptr->ptr->ptr->i`	`i`

Aside from building the data flow chains previously described, data flow performs several additional functions. The locations of all `break`, `continue`, and `return` statements in parallel code are collected, because they may need additionally attention when we perform transformations on the parse tree, but we cannot tell at this point. Data flow collects the locations of all member functions called from parallel code. Data flow for assignment operators also merits special discussion.

Assignment operators in Dataparallel C are heavily overloaded. Data flow attempts to determine at compile time the correct meaning of assignment operators in parallel code. First it checks the shape of the left hand expression of the assignment operator. If the shape is an array and not a single item, then the right hand side must also be an array or pointer. This being true, the assignment node is marked as an array assignment which will be unparsed differently when the final C code is emitted. Next, the mono/polyness of the assignment operands are determined where the possible types are:

Strictly Mono – the expression references a single instance of a variable, and the determination of which variable does not require being evaluated in parallel code.

Loosely Mono – the expression references a single instance of a variable, and the particular instance does require being evaluated in parallel code, such as `mono[+= 1]` since it will evaluate to the same instance of data for all virtual processors, but requires evaluation in parallel code for the += reduction operator.

Strictly Poly – the expression references a different instance of data for all virtual processors. Also, the base object is a poly data item or a `this` pointer.

Loosely Poly – the expression references a potentially different instance of a data item, but the base object is of type mono. This is the case where the instances may all be unique or they may not, such as `mono[poly]`.

After typing the operands to the assignment operator, actions are taken depending on the types of both operands.

Mono = Mono, *Regular Assignment*

If both operands specify a mono expression, either strict or loose, the assignment needs no special action. It is marked as a normal mono = mono assignment and data flow continues.

Mono = Poly, *Selection Assignment*

This is a selection assignment where only one poly value will end up being assigned into mono. This may require special attention. The compiler can generate code to perform this assignment deterministically or nondeterministically, as specified at compile time. The node is marked as a selection assignment and added to a list so that it can be handled during the code transformation phase.

Poly = Mono, *Broadcast Assignment*

Since the right hand side evaluates to a single value, no special action is needed. The same value will be assigned into all the pollies, either strict or loose. If the left hand side is loosely poly, and two or more virtual processors reference the same instance, they will simply assign the same value to that instance.

Strictly Poly = Poly, *Broadcast* or *Regular Assignment*

No special action is needed, because the left hand side will evaluate to a different instance for every virtual processor, even though the right hand side may not.

Loosely Poly = Poly, *Multi-Selection Assignment*

The assignment will require special attention, because more than one virtual processor may be assigning different data into the same data instance, but we cannot tell at compile time. The node is marked as a multi-selection assignment, and added to a list so it will be correctly transformed during the code transformation phase.

For the following assignment operators, `op=` means any of the shorthand assignment operators, such as +=.

Mono `op=` Mono, *Synchronized Assignment*

> In the case of an op= assignment operator, special attention will be required, because all virtual processors must evaluate the left hand side as a rval before any evaluate it as a lval. The node is marked as a synchronized assignment node and added to a list to be handled during the code transformation phase.

Mono `op=` Poly, *Binary Reduction*

> This is a reduction assignment operator. The right hand side will first be reduced and then combined with the mono data. This will require the appropriate code transformations. The node is marked as a binary reduction node, and added to a list to be handled later. The mono operand can be either strict or loose.

Strictly Poly `op=` Mono, *Broadcast Assignment*

> This assignment is not a problem since the left hand side will evaluate to a different data instance for all virtual processors.

Loosely Poly `op=` Mono, *Multi-Reduce*

> This is a multi-reduce assignment operator, and will require special attention, because the `op=` operator must be executed sequentially to prevent information from being lost due to overlapped reads and writes of the left hand expression. It is marked as a multi-reduce to be handled during code transformations.

Strictly Poly `op=` Poly, *Broadcast Assignment*

> Broadcast assignment is not a problem, because the left hand side will evaluate to a unique data instance for all virtual processors.

Loosely Poly `op=` Poly, *Multi-Reduce*

> This assignment is similar to the loosely poly `op=` mono, and is also a multi-reduce. It is handled in the same way.

In addition to determining the above information and taking the appropriate action, regular use-def, def-use, and def-def data flow must be computed for both operands to the assignment operator.

Sometimes the compiler is unable to determine if an expression is strictly or loosely poly. In these cases, it must assume loosely poly, but from examining the description of assignment operator data flow above and the code transformations in section 7.4, we see that better code can be generated when it is known at compile time we are working with strictly poly expressions. This is an important reason for using communication macros wherever possible, since the compiler can easily determine that all communication macro references are strictly poly, and generate the more efficient code than if the programmer had written his own communication macros using either pointers or perhaps the `?:` operator and pointer arithmetic on `this`.

6 Constructing The Minimal Sync Set

After performing data flow analysis we know all the data dependencies that must be preserved with barrier synchronizations. Previously we stated that barrier synchronizations were an expensive operation on a MIMD machine. There are several reasons they are expensive and should be minimized. The first is because all physical processors must participate, and on the Sequent Balance and Symmetry computers they introduce a sequential component into the program every place they are used, that scales linearly with the number of physical processors being used (this is an artifact of using the system barrier synchronization routines, but since the Sequent machines have relatively few processors the difference between using the current method and a tree type method would at best be minimal, and perhaps even worse). The reason all physical processors must participate is because in most cases we cannot determine at compile time which virtual processor will be active at the point of the dependency, and therefore we cannot tell what physical processors must be synchronized. This is actually not a serious problem, because usually the dependencies involve enough of the virtual processors that all the physical processors are involved anyway. The benefits from having only partial participation in barrier synchronizations would be minimal at best, because we do no dynamic load balancing (reasons for this are explained when we discuss how domain selects are emulated in the unparse section), and if physical processors were allowed to proceed past others they would most likely run into other barriers that involved the processors that were hanging back. Other reasons we wish to avoid barrier synchronizations is that they reduce the grain size of the parallel code and increase the relative cost of virtual processor emulation, and by doing so limit the speedups we can expect.

Earlier we discussed how multiple overlapping data dependencies could be preserved with a single barrier synchronization placed at any point in the intersection of the nodes they span. If it were not for loops, determining the minimum set of barrier synchronizations for a program would be almost trivial, but loops can cause dependencies to loop back and be more difficult to extract a minimal set from. Quinn *et al.* in [24] describe an algorithm called *SelectSyncs* that given a set of forward and backward dependencies, will find the minimal set of barrier synchronization points that will preserve all data dependencies. While the *SelectSyncs* algorithm does compute a minimal set of necessary barrier synchronizations, it does not necessary insert these synchronizations at optimal places in the code, such as between statements instead in inside expressions.

We have modified *SelectSyncs* to work with dependency spans. Where the original *SelectSyncs* algorithm returned a set of sync points corresponding to the nodes in the parse tree to synchronize at, our algorithm maintains spans corresponding to the intersections of the original data dependency chains. A span includes all nodes that are executed between and including the endpoints. We linearize the parse tree to provide an explicit ordering to all the nodes in the tree. Our linearization represents only one of possibly many orderings, since most expression operators in C do not specify an evaluation order for their arguments. A barrier synchronization at any point in each resulting span is sufficient to preserve the dependency. After we compute the minimal set of spans required, we can then determine the best place in each span to insert the synchronizations. Selecting the sync point in a span is accomplished by traversing through each span, and evaluating the relative cost of adding a barrier synchronization at each node. The node with the lowest cost is then chosen. These costs have been assigned by carefully evaluating what happens to the resulting code when barrier synchronizations are inserted after different node types. It is important to realize, we have chosen these costs ourselves statically, and incorporated them into the compiler. For example, the cost of synchronizing after a '+' node is much higher than a '`;`' node. Also, the cost of synchronizing between the `array` and `[exp]` nodes of `array[exp]` is higher than synchronizing after the entire array reference.

Determining the set of minimal synchronization spans and selecting optimal points in these spans can be viewed as an optimization, since its only function is to improve performance. After data flow we could blindly insert barrier synchronizations at any point in each of the dependency spans and the program would be correct, however since one of our primary goals is the *efficient* translation of Dataparallel C, we consider this a necessary step in the translation process.

After computing the minimal sync set the compiler has all the information it needs to translate the Dataparallel C program into a C program with calls to parallel memory and process management library routines.

7 Code Transformations

The first transformation applied to the Dataparallel C parse tree is moving all variables accessed in parallel code into shared memory. Moving these variables is necessary, because variables accessed inside domain selects will need to be accessible by multiple physical processors. We can easily determine if a variable needs to be placed in shared memory from the data flow information collected. All local and global variables used or defined in parallel code must reside in shared memory. The Sequent systems only support shared global variables. When a variable is moved to shared memory, we rename it in its declaration and all uses to prevent naming conflicts. If the variable was a local variable, it is now a global variable. Normally this is not a problem, but could cause problems with recursion unless the programmer is very careful. Function arguments referenced in parallel code are another problem, because they too must be in shared memory. Our solution has been to declare global shared variables of the same type, and insert code into the beginning of the function that copies all arguments referenced in parallel code into their shared memory counterparts. All uses of these argument variables are then changed into uses of the new shared variables. A problem with this approach is that if the arguments were pointers, the data they point at may or may not reside in shared memory.

Poly variables declared locally in parallel code are handled differently. Each virtual processor must have its own instance. Our solution has been to rename these variables and their uses to avoid naming conflicts, and move the declarations into the domain declaration, making them member variables. This gives every virtual processor its own copy, but also requires potentially large amounts of shared memory.

7.1 Scalarization

In certain cases we are able to apply an important optimization to the movement of local poly variables. If the locally declared poly variable is not live across any barrier synchronization, then it does not need to be moved into shared memory, and can remain a locally declared variable. This is because each virtual processor will be through with it before the physical processor emulates the next virtual processor (since we are using a for loop to emulate virtual processors). The def-use data flow information collected, where the same virtual processor both defines and uses a variable, is used to determine if a local poly variable can be scalarized. Scalarization is an important optimization for two reasons. First, it helps to conserve shared memory, since it allocates only a single variable on each physical processor, instead of one for each virtual processor, and the variable will be allocated on the stack instead of shared memory. Secondly, and perhaps more important, a good optimizing compiler will often allocate these local poly variables in registers which can have a dramatic effect on improving performance. In practice, this may be the most important performance-improving optimization implemented by our compiler.

We call this optimization scalarization, because it is a mechanism by which a poly variable is actually implemented as a single variable. Note that this is not the same transformation as scalar expansion or loop scalarization, performed by some vectorizing compilers as outlined in [19] and [30].

7.2 Domain Invariant Code Motion

A statement is considered *domain invariant* if all contained expressions are strictly mono, and it is either at the top level inside a domain select (not nested in any control structures), or it is only nested in domain invariant control structures. There is no reason a domain invariant statement needs to remain in parallel code. In fact, there are good reasons why they often should not. It is usually cheaper to execute the statement only once instead of once for each virtual processor. Also, moving domain invariant code out of parallel code can reduce the number of barrier synchronizations, divided domain selects, and temporary variables. As an example of why this is true, consider the code:

```
int mono;

[domain foo].{
  mono++;
  statement_block;
}
```

Without domain invariant code motion, this code would be translated as:

```
int mono;

[domain foo].{
  tmp = mono+1;
}

barrier_synchronization

[domain foo].{
  mono = tmp;
  statement_block;
}
```

With domain invariant code motion, the original code becomes:

```
int mono;

mono++;

[domain foo].{
  statement_block;
}
```

The compiler performs domain invariant code motion by traversing through the parse tree looking for statements that meet these criteria. When it locates a domain invariant statement, it divides the domain select immediately before the statement, and then moves the statement in between the two subdivided domain selects.

This optimization is particularly useful for improving code when domain invariant `for` loops have been placed inside parallel code, due to the costly method in which they would be handled otherwise (an example of which is given later, in section 7.4).

Sometimes this optimization does not improve code, but actually worsens it. Consider the code:

```
[domain foo].{
  statement_block1;
  mono2 = mono1;
  statement_block2;
}
```

Lets assume there are no dependencies between `statement_block`$_1$ and `statement_block`$_2$. If we apply domain invariant code motion to this code we would have:

```
[domain foo].{
  statement_block1;
}
mono2 = mono1;
[domain foo].{
  statement_block2;
}
```

If domain invariant code motion had not been applied, we would still have:

```
[domain foo].{
  statement_block1;
  mono2 = mono1;
  statement_block2;
}
```

which is often better code, because there is one fewer domain select and the cost of each virtual processor executing `mono2 = mono1` is lower than the overhead of the extra domain select. If the domain invariant code had been more expensive, this optimization would have actually been an improvement, and it usually is. If the domain invariant statement is either the first or last statement in a domain select, it always improves code, because it introduces no extra domain selects.

Domain invariant code motion is an optimization that can always be performed at the source level by the person coding the program. We expect that experienced Dataparallel C programers would rarely place domain invariant code in a domain select, but would perform the optimization themselves as the code was written.

7.3 Fix up return/continue/break Statements

Any loop in parallel code that contains a continue or break and also requires a barrier synchronization in its body will require special transformations. These transformations will be discussed, as well as why they are necessary when we explain loop synchronization transformations. Parallel returns are only allowed in member functions, and they too require special attention. We present the details for how parallel returns are handled in the code transformation and unparse sections.

7.4 Division of Domain Selects

Since Dataparallel C is a SIMD language, code in a domain select will be executed synchronously from top to bottom for all virtual processors. Therefore, dividing a domain select into multiple domain selects does not change the semantics of the original program. (This may not be true in the presence of variables declared locally to the domain select, but can be made true if these local variables are moved into the domain declaration as a domain member, and possibly renamed in their declaration and all uses to avoid naming conflicts.) For example, the following two code fragments are equivalent.

```
[domain foo].{
  statement_block_1;
  statement_block_2;
}
```

and

```
[domain foo].{
  statement_block_1;
}
[domain foo].{
  statement_block_2;
}
```

There is no requirement that the subdivisions must be between two statements, and in fact they can be almost anywhere as long as the order of execution is preserved. Splitting a single statement or expression is not enough to divide the domain select. The process must be continued up through the parse tree until the domain select is finally divided. For example, if an expression is split, then the statement containing it must be split. Next, any statements containing that statement must be split. This continues until the domain select itself is split.

When a statement or expression is split, there are two resulting code segments, a *before* synchronization segment and an *after* synchronization segment. The after synchronization segment is constructed in the existing parse tree, while the before synchronization segment is maintained in a separate new parse tree. As splitting progresses up the parse tree toward the surrounding domain select, the before segment is passed up the tree as well. At each step, the split operation is performed on the surrounding statement or expression, possibly modifying the after code in place and adding to the before parse tree. The action taken depends on three factors. The first is the kind of node we are splitting (`if` statement, expression statement, binary operator, etc.). The second is the state of the before tree being passed up the tree; if it is empty we are just initiating a split, otherwise we are in the process of splitting up the tree towards the current domain select. If we are in the process of splitting up the tree and the current node has multiple children, the last determining factor is which child node did we traverse up from (this can be very important in determining what should go in the before tree and what goes in the after tree). When the domain select is finally reached, the before parse tree will contain all the code that must be executed before the barrier synchronization, and only the code to be executed after the synchronization will remain in the existing domain select. The before parse tree is then wrapped in a new domain select, and this new domain select is inserted immediately before the existing domain select. A simple example of the splitting is shown below. In the following examples `[`*sync*`]` represents where the barrier synchronization needs to be placed.

Initial code:

```
Before:   NULL                        After:   [domain foo].{
                                                 statement_1;
                                                 statement_2;
                                                 [sync]
                                                 statement_3;
                                               }
```

After splitting one node:

```
Before:   statement_2;                After:   [domain foo].{
                                                 statement_1;
                                                 [sync]
                                                 statement_3;
                                               }
```

After splitting two nodes:

```
Before:   statement_1;                After:   [domain foo].{
          statement_2;                           [sync]
                                                 statement_3;
                                               }
```

After splitting three nodes:

```
Before:   {                           After:   [domain foo].
            statement_1;                       [sync]
            statement_2;                       {
          }                                      statement_3;
                                               }
```

Finally, after splitting the domain select:

```
Before:   [domain foo].{
            statement_1;
            statement_2;
          }
          [sync]
After:    [domain foo].{
            statement_3;
          }
```

We have developed a set of transformations to split all expression and statement types. How these are implemented by our compiler is too complex to explain fully here, but they can be viewed as a series of parse tree rewriting rules, and we will present them as such. We now describe these transformations.

7.4.1 Expression Transformations

If the barrier synchronization must be placed inside an expression, then we rewrite the expression as two separate expressions (a *before* and *after* expression). This requires creating a temporary poly variable to hold the result of the before part of the expression, to use while evaluating the after part of the expression. Since the value of the before expression could be different for every virtual processor, all virtual processors will need their own temporary, placed in the domain declaration so that it will remain live across the barrier synchronization (in the case where the before expression evaluates to a mono value, a single shared global variable could be used, but we have not implemented this optimization). It is possibly that more than one expression transformation will need to be applied to successfully split the expression. After completely splitting the expression into two statements, statement transformations will need to be applied until the domain is completely split.

General Expression Transformation

If an expression split is just starting (this can be determined if before is null), the entire subtree rooted at the current node must be evaluated before the rest of the expression tree the subtree is embedded in, and a barrier synchronization enforced between them. We create a new temporary as a member in the current domain type and replace the subtree with a reference to this temporary. The type of the temporary is determined from the type of the subtree being replaced. Type information has been cached in the parse tree during the parse phase. An assignment operator is then created, assigning into the new temporary the subtree that was unhooked. The new expression tree is then bound to before. Before is passed up the tree, and splitting continues until the domain select is split.

If the expression split is in progress (before is not null), and the current node is an expression node, then the usual action is to pass before on up the parse tree. The exceptions to this are those operators that specify the order of evaluation for their operands; the `&&`, `||`, `?:`, and comma operators. These are examined in detail later.

As an example of the expression splitting process, consider the expression

```
var1 = exp1 + exp2 * exp3 *(S) exp4 + exp5;
```

Which has as a parse tree

```
      =
    /   \
var1     +
       /   \
      +     exp5
    /   \
exp1     *(S)
       /   \
      *     exp4
    /   \
exp2 exp3
```

Initially before is null. A barrier synchronization is required after evaluating the node marked with an (*S*). After the initial transformation we have

```
Before:     =                        After:     =
          /   \                               /   \
      tmp1     *                          var1     +
             /   \                               /   \
            *     exp4                          +     exp5
          /   \                               /   \
      exp2  exp3                          exp1    tmp1
```

Now splitting continues up the parse tree, which merely passes the before subtree up, giving

```
Before:  this->tmp₁ = exp₂ * exp₃ * exp₄
         [sync]
After:   var₁ = exp₁ + this->tmp₁ + exp₅
```

In the example, each of the *exp* nodes could be a complex subtree or as simple as an id or literal reference. After the expression tree is completely split, statement splitting will be required to split the domain select.

The general expression transformation may seem to be incorrect for certain possible unary expression splits, such as splitting `&base` between the reference to base and the address of operator, or splitting `++var` between the ++ operator and the `var` reference, and indeed they are. Our transformations would yield

```
tmp = base, [sync] &tmp
```

for `&[sync]base`, and

```
tmp = var, [sync] ++tmp
```

for `++[sync]var`, both of which are errors. What makes these transformations work is that the compiler would never try to synchronize immediately before the unary op, because the synchronization could always be moved up higher in the parse tree for a less costly synchronization (after the address of operator or ++ operator had been evaluated) without effecting the data dependency. If node splitting ever traverses up to a unary operator, it will always have traversed up from a subexpression of the argument to the unary op, such as the array subscript expression in `&array[exp]`.

Unary ops are split as described above except when using a post or pre increment or decrement operator on a mono variable in parallel code. A barrier synchronization will be forced at the increment or decrement node. Before will be null, the unary node type will be either a pre/post increment or decrement operator, and the argument to the operator a mono variable reference. Special action must be taken, because all virtual processors must evaluate the argument first as an rval, synchronize, and then evaluate it as an lval; because the value of the argument is only adjusted once, and not once for every virtual processor. This is a transformation that yields expensive code, and gives good reason why for loops with mono iterators should be avoided in parallel code wherever possible.

The examples below shows how pre and post increment and decrement operators are transformed when their argument is a mono variable reference:

```
++mono                                   mono--
```

The parse trees are initially

```
        ++(S)                                --(S)
          |                                    |
        mono                                 mono
```

and before is null. We create a poly temporary in the current domain type, of the same type as the mono expression. We then replace the subtree including the ++/– operators with a assignment of the temporary plus or minis one into the mono variable. A new tree that assigns the value of mono to temp is created and bound to before, resulting in

```
Before:    =         After:     =            Before:    =           After:        ,
          / \                  / \                     / \                       / \
       tmp   mono          mono   +                 tmp   mono                  =   tmp
                                 / \                                           / \
                              tmp   1                                      mono   -
                                                                                 / \
                                                                              tmp   1
```

or

```
Before:   tmp = mono           tmp = mono
          [sync]               [sync]
After:    mono = tmp + 1       mono = tmp - 1, tmp
```

Splitting then proceeds up the parse tree.

The primary difference between how the pre and post operators are transformed is the post operators use the comma operator in the after expression as a mechanism to return the value of the original mono variable before it is modified (the value of `tmp`).

Special Expression Node Types

The `&&`, `||`, `?:`, and comma operators must be handled specially, because they all enforce a strict order of evaluation on their operands (left to right). Also `&&` and `||` do not always evaluate both of their operands in the case of a short circuit. The `?:` operator always evaluates only two of its three operands. While it may seem reasonable to rewrite the `&&`, `||`, and `?:` operators in terms of an `if` statement, the transformations always rewrite each one in terms of themselves.

If an initial split must be performed on one of these operators, the general expression transformation may be used. If the initial split occurred somewhere in the leftmost operand subtree, we can just pass before up the parse tree, because each of these node types always evaluates its first expression, and always evaluates it first.

When splitting a comma operator, and splitting has propagated up from the rightmost operand, the transformation must ensure that the left operand is evaluated before anything in the before tree. The transformation used is to replace the comma operator subtree with its right operand subtree, and to replace the right operand of the comma operator with the before subtree, and then bind this new comma operator to before. As an example consider the expression where splitting is already in progress, and exp_2 has just been split creating exp_4:

Before: exp_4 After: exp_1, exp_2, exp_3

The expression exp_4 would assign into a temporary that was later used in exp_2. The parse trees for the before and after expressions are:

```
Before:  exp4                    After:      ,
                                            / \
                                           ,   exp3
                                          / \
                                      exp1 exp2
```

When splitting propagates up to the first comma op, from the right operand, the transformation yields:

```
Before:      ,                   After:      ,
            / \                             / \
        exp1   exp4                     exp2   exp3
```

This will ensure that the order of evaluation for exp_1 and exp_2 remain correct. Splitting then propagates up to the last comma op, but from the left operand. The order of evaluation is already correct with no additional transformations, so splitting will just propagate up to the next level. The results would be:

Before: exp_1, exp_4 After: exp_2, exp_3

When splitting propagates up to a `||` or `&&` operator from the right subtree, a similar transformation must be applied that will cause before only to be evaluated if the value of the left operand requires it. This can be accomplished as shown in the example:

Before: exp_3 After: exp_1 `||` exp_2

Now, exp_3 would assign into a temporary that was later used in exp_2. The parse trees for the before and after expressions are

```
Before:  exp3                    After:     ||
                                           / \
                                       exp1 exp2
```

When splitting propagates up to the `||` op from the right operand, the transformation yields:

```
Before:        ||                After:     ||
              /  \                         / \
            ()    exp3                 tmp2 exp2
             |
             =
            / \
        tmp2   exp1
```

The final result is

```
Before:    (tmp₂ = exp₁) || exp₃
           [sync]
After:     tmp₂ || exp₂
```

This preserves both the order of evaluation and the optional argument evaluation properties of the `||` operator. The `&&` operator is treated identically but using `&&` in place of `||`. A similar transformation exists for the `?:` operator.

Reductions (both unary and binary), selection assignment, and multi-reduce assignment operators pose a special problem. The transformations we have chosen for performing reductions are to have each physical processor first perform a local reduction for all the currently active virtual processors it is emulating. When each physical processor finishes its local reduction, it drops out of the virtual processor emulation loop, locks the bus, reduces into the global variable, unlocks the bus, and proceeds. Locking the bus is done to ensure mutual exclusion. For unary reductions, the global variable is a temporary that is created, and pre-initialized. For binary reductions, it is the left-hand argument being reduced into, which will be a mono variable. Local reductions are performed first, because only order #PEs locks will be required, instead of order #VPs locks. (Locks create a sequential component, and therefore should be avoided where possible.) Examples of the code generated by reductions is given in the code generation section.

A selection assignment can have only one winner. The language specification leaves open the issue of which virtual processor will win, in the case of multiple virtual processors attempting to perform the assignment. We have implemented two models, selectable at compile time. The first is a nondeterministic model in which it is arbitrary which virtual processor will make the last assignment, which will be the winning assignment. We implement this by having each virtual processor make the assignment from parallel code into the mono variable. If the variable is larger than an atomic bus transfer (32 bits for the Sequent's), such as a double, struct, or array, we must lock the bus before each assignment, and unlock it after. This creates a sequential component in the program of order #VPs.

The other model we have implemented is a deterministic model. Here, the virtual processor that successfully makes the assignment will be the lowest numbered active virtual processor, where lowest numbered means the active virtual processor with the lowest value for `this`. The transformation for a deterministic selection assignment creates a mono variable that is initialized to infinity before the current block of parallel code is entered. Each active virtual processor first checks to see if its value for this is smaller than the global mono variable. If it is, it locks the bus, and rechecks to see if its value of this is still lower than the current global this value, and if so it assigns its value of `this` to the global minimum `this` variable and performs the selection assignment. The bus is then unlocked. Since each physical processor emulates its share of virtual processors in order of increasing `this`, they will never perform more than one lock, and maybe none. The sequential component for the deterministic selection assignment is of order #PEs, and measurement has shown that the deterministic selection assignment usually outperforms the nondeterministic selection assignment. A complete example of the code generated for selection assignments is given in the code generation section.

The transformation for multi-reduce is identical to the nondeterministic selection assignment. This means we have no deterministic multi-reduce when the reduction operator is a simple assignment, such as `mono[poly] = poly`.

Member Function Invocations

As mentioned in the parse and scan section, member functions provide their own virtual processor emulation. However, they still expect to be called with all physical processors active (multi mode). Since they provide their own virtual processor emulation, they cannot be invoked from within virtual processor emulation. What is required is to split the domain select, and to place the function call between the domain selects. There are two problems caused by this scheme, how to handle arguments for the member function, and how to handle the return value.

We have handled both these problems by adding member variables to the domain type. All the member function arguments are moved into the domain as members. A new member of the return type for the member function is created in the domain to return the function value through. (These member variables are only created once, during the scan and parse phase. It is mentioned here, because it is easier to understand than if it was presented earlier.) Before the member function is called, assignment statements are inserted into the domain select that initialize the argument member variables to the values being passed to the member function. In the split domain select after the call to the member function, the return member variable is used for the return value of the member function. An example will help clarify the procedure.

Untransformed Code:

```
domain foo.{
  int i, j;
  float baz(int a, int b);
}

...

[domain foo].{
  statement_block1;
  i = baz(i, 3)/j;
  statement_block2;
}
```

Transformed Code:

```
domain foo.{
  int i, j;
  float baz();
  int baz_arg_a, baz_arg_b;
  float baz_return;
  int baz_active;
}

...

[domain foo].{
  statement_block1;
  baz_arg_a = i;
  baz_arg_b = 3;
  baz_active = 1;
}
/* stay in multi mode here! */
baz();
[domain foo].{
  i = baz_return/j;
  statement_block2;
}
```

In the example above, `baz_active` is the virtual processor active variable that was discussed in the scan and parse phase. It must be initialized to true for all virtual processors calling the member function, and false for all those not.

7.4.2 Statement Transformations

Both inter-statement and intra-statement barrier synchronizations will be required during the transformation process. An initial synchronization transformation could be between two statements, or the splitting process could propagate a synchronization up to a statement from some point internal to the statement.

Most examples for statement transformations will be presented as source to source transformations, instead of parse tree transformations, because this makes them easier to understand, and the extra insight provided by the parse trees is not necessary. In the examples, before in the untransformed code corresponds to the before parse tree at the start of performing this transformation (what was passed up from below). In some instances before will be an expression, and in others a statement. This should be clear from the context of its use in each example. In the transformed code, everything before [*sync*] corresponds to before after performing the transformation, and will be passed up the parse tree. Everything after [*sync*] is left in the existing parse tree in place of what was in the untransformed parse tree. We now discuss these statement transformations as implemented by the compiler.

General Statement Transformation

The general statement transformation is very similar to the general expression transformation. The major difference is no temporaries need to be created, or assignment expressions created. What is done it to unhook the subtree rooted at the current node, and replace it with an empty expression statement node. Before is bound to the subtree that was unhooked, and passed up the parse tree to continue the splitting process.

As a simple example consider

```
statement_1;
statement_2;
[sync]
statement_3;
```

Which has as a parse tree

```
     sl
    /  \
st_1;   sl
       /  \
(S) st_2;  st_3;
```

The node labeling `sl` stands for statement list. Initially before is null. A barrier synchronization is required after evaluating the node marked with an (*S*). The initial transformations gives

```
Before: statement₂;                 After:     sl
                                              /  \
                                          st₁;   sl
                                                /  \
                                               ;    st₃;
```

Now splitting continues up the parse tree, and additional transformations will be applied.

Return and Expression Statements

If splitting traverses up to an expression or return statement from an expression split, all that must be done is to turn before from an expression into an expression statement, and the splitting process can continue to traverse up the parse tree. This effectively transforms the original statement and expression into two equivalent statements that will have a barrier synchronization between them.

We show examples of splitting both return and expression statements, where the original expression has already been split into before and after expressions:

Untransformed Code: Transformed Code:

```
before_exp [sync] after_exp;                   before_exp;
                                               [sync]
                                               after_exp;

return before_exp [sync] after_exp;            before_exp;
                                               [sync]
                                               return after_exp;
```

Statement List

A statement list is a group of statements in the body of a compound statement. The transformation is very similar to that of the comma operator. For an initial split, the general statement transformation is used. If it is not an initial statement split, we have traversed up from either the left or right subtrees. The statement list specifies an order of evaluation which must be preserved; the left subtree must be evaluated before the right. If we traverse up from the left subtree, the order is still intact, and we can continue the upward traversal. If we traverse up from the right subtree, we must ensure that everything in the left subtree is evaluated before anything bound to before. We create a new statement list with the left subtree of the current node as the left subtree of the new statement list, and before as its right subtree. The new statement list is then bound to before. The current node is then replaced with its right subtree.

Continuing the example for the general statement transformation, we would traverse up to the lower statement list node from the left subtree, so no transformation would be performed, and we could continue to traverse up the parse tree. The next traversal would again take us to a statement list, but this time from the right subtree. After applying the necessary transformation we would have:

```
Before:   sl                        After:     sl
         /  \                                 /  \
      st₁; st₂;                              ;    st₃;
```

These parse trees correspond to

```
Before:  statement_1;
         statement_2;
         [sync]
After:   statement_3;
```

From this point, splitting continues up the tree.

Compound Statement

If during the splitting process we traverse up into a compound statement, the before tree is wrapped in a compound statement. We know that no local variables will be alive across the barrier synchronization, because they would have been moved to the domain declaration during the variable movement transformation. However, they may be needed in both the before and after code segments, so any declarations in the preexisting compound statement are duplicated in the new compound statement. Splitting then traverses on up the parse tree.

Untransformed Compound Statement:

```
{
  type var_1, var_2;
  before_statement_1;
  [sync]
  statement_2;
}
```

Transformed Compound Statement:

```
{
  type var_1, var_2;
  before_statement_1;
}
[sync]
{
  type var_1, var_2;
  statement_2;
}
```

If Statement

Splitting can traverse up into an `if` statement from three possible subtrees, the test condition, the true code block, and the false code block. These three cases must be handled separately.

Case 1 – If splitting has traversed up from the test condition, the before expression must be converted into an expression statement. Before can then be passed on up the parse tree and splitting continued. The has the effect shown below:

Untransformed `if` Statement:

```
if (before_exp [sync] exp) {
  statement_block;
}
```

Transformed `if` Statement:

```
before_exp;
[sync]
if (exp) {
  statement_block;
}
```

Case 2 – If the `if` statement was traversed into from the true code body, then the `if` statement will be split into two `if` statements with a barrier synchronization between them. To do this, the compiler generates a poly temporary, assigns it the value of the test condition, and inserts this assignment statement in the parse tree immediately before the `if` statement. The `if` statement test condition is then replaced with a reference to the poly temporary. When the `if` statement is split into two `if` statements, they will both have a reference to this temporary as their test condition, and the original test condition will only be evaluated once. This prevents the possibility of side effects involving the test condition between the before and after `if` statements. We create a before `if` statement with what was bound to before at the start of the `if` statement split as its true code body. Its false code body will be null.

Untransformed `if` Statement:

```
if (exp) {
  before_statement_block;
  [sync]
  statement_block2;
} else {
  statement_block3;
}
```

Transformed `if` Statement:

```
tmp = exp;
if (tmp) {
  before_statement_block;
}
[sync]
if (tmp) {
  statement_block2;
} else {
  statement_block3;
}
```

Case 3 – When an `if` statement is traversed into from the false code body, the transformation is identical to case 2, except that the new before `if` statement will have the original `if` statement's true code body for its true code body, and what was originally bound to before for its false code body. The after `if` statement will have a null true code body. This transformation preserves the order of execution for the true and false code bodies of the `if` statement.

Untransformed `if`:

```
if (exp) {
  statement_block1;
} else {
  before_statement_block;
  [sync]
  statement_block3;
}
```

Transformed `if`:

```
tmp = exp;
if (tmp) {
  statement_block1;
} else {
  before_statement_block;
}
[sync]
if (tmp) {
/* Null Compound Statement */
} else {
  statement_block3;
}
```

Switch and Label Statements

The synchronization transformations for a `switch` statement are identical to an `if` statement with no else clause. However, if the body of a switch requires a barrier synchronization, the body will undergo some unique transformations, because of the `case` and `default` statements that may require splitting.

Both case and default are label statements, and are handled identically. When the body of a switch statement is split, the order of the code bodies for the label statements must be preserved. When splitting traverses up to a label statement, or starts at a label statement, the label statement is moved up to the top of before, and a new label statement with an identical expression is left in its place. If the code in the case or default contained a break statement, a break statement is added as the code body for the new label statement, otherwise it has a null code body. The example below helps to clarify how the transformation is accomplished.

Untransformed switch Statement:

```
switch (exp$_1$)
{
  case 1: statement_block$_1$;
          break;
  case 2: statement_block$_2$;
  case 3: statement_block$_3$;
          [sync]
          statement_block$_4$;
  case 4: statement_block$_5$;
  default: statement_block$_6$;
}
```

Transformed switch Statement:

```
tmp = exp$_1$;
switch (tmp)
{
  case 1: statement_block$_1$;
          break;
  case 2: statement_block$_2$;
  case 3: statement_block$_3$;
}
[sync]
switch (tmp)
{
  case 1: break;
  case 2:
  case 3: statement_block$_4$;
  case 4: statement_block$_5$;
  default: statement_block$_6$;
}
```

This example shows two reasons why identical label statements must be duplicated in the after code. First, to prevent virtual processors for which tmp is either one or two from executing the default code in the switch statement after the barrier synchronization. Secondly, virtual processors with tmp equal to two should fall through into *statement_block*$_4$. Our transformation scheme achieves both these criteria.

The Efficient Translation of Synchronized Loops

Because of the large amount of time that tends to be spent in program loops, efficient loop transformations are important.

When a looping construct requires a barrier synchronization either in its body or a controlling expression, the loop must be rewritten to bring the barrier synchronization out of the virtual processor emulation loop. This is necessary so that all virtual processors will have executed the code prior to the barrier synchronization before any execute the code after the barrier synchronization (remember, SIMD semantics here!).

The objectives of the loop transformations are to first, provide a correct translation that will behave with the desired semantics, and secondly to translate the loop in such a way as to incur as little additional overhead as possible. These goals are achieved by using a translation scheme that (1) places as few barrier synchronizations as possible in the body of a loop, especially innermost nested loops, even at the expense of placing extra syncs outside the loop or in outwardly nested loops; (2) avoids a global reduction each time through the loop to determine if any virtual processors are still executing the loop (remember, all physical processors must participate in every barrier synchronization, so all processors must continue performing barrier synchronizations as long as any processor has virtual processors that are still looping); (3) avoiding switching from multi mode to single mode and back to multi mode in the loop body; and (4), reducing contention for locked variables which are necessary for determining when all processors are done with a loop.

If a loop needs no barrier synchronizations in its body or test expression, it undergoes no transformations, and is emitted identically to how it was written.

For a `for`, `while`, or `do-while` loop requiring a barrier synchronization in its body, test expression, or increment expression, the following methods are used.

If the loop is a `for` loop, it is first transformed into a `while` loop by moving the initialization expression immediately before the loop, and the increment expression is moved to the last statement in the body of the loop. (To be more precise, a new compound statement is created with the original `for` loop body as the first part, the increment expression as the last part. In this way aliasing of any local variables declared in the loop body and any variables used in the increment expression, is not a problem.) By moving these expressions the loop can be switched from a `for` to a `while` loop. The `while` loop transformations can then be applied.

Let us examine the case of a simple `while` loop with a single barrier synchronization required in its body, such as

```
[domain foo].{
  statement_block1;
  while (test_exp)
     {
        statement_block2;
        [sync]
        statement_block3;
     }
  statement_block4;
}
```

We can transform the loop as shown:

```
VP_EMULATION_LOOP {
  statement_block1;
  poly_is_vp_active = 1;
}
/* no barrier synchronization needed here!  Can stay in multi mode! */
```

The code above executes `statement_block1` normally, and then each virtual processor initializes its value of `poly_is_vp_active` to 1. The variable `poly_is_vp_active` is used in the transformed loop by each virtual processor to determine if it is still executing the loop.

```
{
  int local_mono_is_PE_active;

  global_mono_numprocs_looping = numprocs;
```

Every physical processor declares its own local variable local_mono_is_PE_active that is used to determine if any of the virtual processors being emulated by a physical processor are still active. The variable global_mono_numprocs_looping is a single shared global variable that is initialized to the number of physical processors that still have active virtual processors executing the loop. When it goes to zero and becomes false, the loop is finished. The use of these variables helps avoid a global reduction each time through the loop to determine if any virtual processor is still executing the loop.

```
do
{
  local_mono_is_PE_active = 0;
```

We now are in the transformed loop. Note that we are not emulating virtual processors at this point, but all physical processors are active. Every processors sets its local_mono_active variable to zero each time a loop iteration is begun. It is only set back to 1, below, if a processor determines it still has active virtual processors.

```
VP_EMULATION_LOOP {
  if (poly_is_vp_active) {
   poly_is_vp_active = test_exp;
   if (poly_is_vp_active) {
     local_mono_is_PE_active = 1;
     statement_block2;
   }
  }
}
```

The virtual processor emulation loop first tests poly_is_vp_active for each virtual processor, and if it is one (meaning active the last time through the translated loop, or if this is the first time through the loop), it re-evaluates the test expression and assigns the new value to poly_is_vp_active. If poly_is_vp_active was 1, and is still 1 after evaluating the test expression, the virtual processor sets local_mono_is_PE_active to 1, signaling that the physical processor still has active virtual processors (this is a local reduction instead of a global reduction, so it avoids any critical sections), and then executes *statement_block*$_2$. This is repeated for all active virtual processors.

```
m_sync();
```

At this point we reach the barrier synchronization, and all physical processors wait until they are synchronized. All virtual processors will be done with *statement_block*$_2$ before we proceed.

```
VP_EMULATION_LOOP {
  if (poly_is_vp_active) {
   statement_block3;
  }
}
/* no barrier synchronization, stay in multi mode */
```

We now enter a virtual processor emulation loop for $statement\_block_3$. Any virtual processors that were active above will execute $statement\_block_3$. There is no need to synchronize after this virtual processor emulation loop.

```
} while (local_mono_is_PE_active)
```

Each physical processor tests its own local copy of `local_mono_is_PE_active` to determine if any of its virtual processors need to continue looping. Only physical processors that still have active virtual processors will continue to execute the loop; however, all physical processors will continue to participate in the barrier synchronizations through the mechanism described below.

```
if (num_syncs & 1) m_sync();
```

Since the number of barrier synchronizations in the above loop was odd, physical processors could drop out of the loop after executing either an even or an odd number of barrier synchronizations, depending on the number of trips made through the loop. For the code below, it is important that all physical processors have executed either an even or an odd number of barrier synchronizations, but not both. Here we have arbitrarily chosen even. The `if` statement above tests the number of barrier synchronizations, and if it is odd, adds an extra barrier synchronization so it will be even. It is important to realize here, that `numsyncs` is the number of barrier synchronizations executed in the program so far, and not just in the loop. This becomes important when transforming nested loops.

```
lock();
global_mono_numprocs_looping--;
unlock();
```

When each physical processor finishes the loop, the `global_mono_numprocs_looping` variable must be decremented (the count of the number of physical processors still executing the loop). It is a global shared variable, so this is a critical section and the bus must be locked, since multiple processors could be executing this code simultaneously.

```
    for (;;) {
      m_sync();
      if (!global_mono_numprocs_looping) break;
      m_sync();
    }
}
```

In the final `for` loop above, all physical processors are executing exactly out of sync with the decrement operation above. This is guaranteed by the `if`, because the decrement is only executed when the number of barrier synchronizations is even, and there are two barrier synchronizations, one for even syncs and one for odd. In this way all the physical processors test the `global_mono_numprocs_looping` variable in the `if` statement only after an odd number of barrier synchronizations, which guarantees that it will not be adjusted by a processor above until after another barrier synchronization, in which case no processors will be testing it. When `global_mono_numprocs_looping` goes to zero, all physical processors have finished the transformed `while` loop and they exit the `for` loop, thus ending the original transformed `while` loop.

There are other simpler translation schemes, but they have the disadvantages of requiring a global reduction in the main loop, a two-sync minimum in the main loop, switching from multi to single and back again, or all three. The scheme presented avoids all of these, but does require potentially two extra barrier synchronizations at the end of the loop. In most circumstances this will be much faster than having an extra barrier synchronization executed every time through the loop.

As long as the number of barrier synchronizations in the loop is odd, the above translation can be used. If it is even, a small optimization can be made. Since it will be known upon exiting the `do-while` that there have been an even number of barrier synchronizations, the

```
    if (num_syncs & 1) m_sync();
```

statement can be completely eliminated. This reduces the number of extra barrier synchronizations from two to one. There may be other cases where at compile time it is not possible to determine whether the number of barrier synchronizations in the loop body is odd or even, such as when a member function is called. In these cases the odd sync test cannot be eliminated.

In the case of nested loops, the translation scheme is slightly more complex. Each nested loop needs its own global shared variable for the number of active physical processors which must be initialized to the surrounding loops shared number of active processors variable each time before entering the inner loop. This initialization creates another problem. If a physical processor has just dropped out of the outer loop, and another has looped back around and is about to start the inner loop, we must guarantee that the processor that dropped out of the outer loop has finished updating the active physical processor count before we use it to initialize the physical processor count for the inner loop. It may take either one or two syncs when a processor drops out of the outer loop before the variable has been updated, depending on if the synchronization `if` is required for the outer loop. If there are at least two barrier synchronizations between the beginning of the outer loop and the beginning of the inner loop, then there is no problem and the outer loop active processor variable may be safely read. If there are not enough barrier synchronizations, then one or two extra barrier synchronizations will need to be added before reading the outer loops active processors count. This is only a potential problem for the first nested loop in the example below, because there will always be enough barrier synchronizations for the second (a loop will always execute a minimum of two barrier synchronizations).

```
while (i) {
  while (j) {
    ...
  }
  ...
  while (k) {
    ...
  }
}
```

In translating an outer loop and nested loops with barrier synchronizations, it is no longer necessary to know if there are an even or odd number of barrier synchronizations in the body of the outer loop or loops, since the last innermost nested loop will only exit on an odd number of syncs, all outer loops will be synchronized by this and will never need the `if` test. (For any given loop, physical processors can exit either on an odd sync count or an even sync count, but never both.)

The process for translating `do-while` loops is very similar, but slightly less complex due to the loop body always being executed at least once.

Break and Continue Fix ups

It can be seen from the above loop transformations that after transforming the loops, any `break` or `continue` statements embedded in the original loop would no longer behave correctly, because the virtual processor emulation loop and the transformed loop have effectively been interchanged. This would cause `break` and `continue` statements to refer to the virtual processor emulation loop, and not the loop they were originally embedded in.

To maintain the original semantics of the Dataparallel C program, we must fix up `break` and `continue` statements in loops that require barrier synchronizations, before the loop is transformed. During the data flow phase of the compiler, the locations of all `break` and `continue` statements are recorded. After the minimal sync set has been computed and before any code transformations are applied, we check each recorded `break` and `continue` to see if it is in a loop that requires a barrier synchronization. If not, the `break` or `continue` is left intact. If it is, the `break` or `continue` must be rewritten in a form that does not use `break` or `continue`, so that it will have the correct semantics after the loop has been transformed.

Continue

A `continue` statement in a loop in parallel code causes all virtual processors that execute a `continue` to immediately proceed to the bottom of the loop. Those virtual processors that do not execute a `continue` statement execute the rest of the loop normally.

If no barrier synchronizations are needed between the `continue` statement and the end of the loop, the `continue` can be replaced with a `goto` statement that jumps to the end of the loop. However, if a barrier synchronization is required between the `continue` statement and the end of a loop, the `goto` cannot be used, because our transformations do not correctly handle `goto`'s spanning barriers. (They do not need to, because to the language restriction prohibiting the use of `goto` in parallel code.) Instead we enclose the remainder of the loop body in an `if` statement, whose condition is a poly temporary we create that reflects whether or not the continue was taken. Adding the necessary `if` statement may require additional transformations, because the `continue` could be nested in conditional code. The `if` type fix up can always be used, but we use the `goto` type fix up whenever possible, because it results in more efficient code. Examples of both types of `continue` fix ups are shown below:

Before `continue` Fix up:

```
while (exp1)  {
  statement_block1;
  [sync]
  statement_block2;
  if (exp2) {
    continue;
  }
  statement_block3;
}
```

After `goto` Fix up:

```
while (exp1) {
  statement_block1;
  [sync]
  statement_block2;
  if (exp2) {
    goto label;
  }
  statement_block3;
label;
}
```

Before `continue` Fix up:

```
while (exp1) {
  statement_block1;
  if (exp2) {
    continue;
  }
  statement_block2;
  [sync]
  statement_block3;
}
```

After `if` Fix up:

```
while (exp1) {
  tmp = 1;
  statement_block1;
  if (exp2) tmp = 0;
  if (tmp) {
    statement_block2;
    [sync]
    statement_block3;
  }
}
```

Break

A `break` statement in a loop in parallel code will cause those virtual processors that execute it to immediately exit the loop. The virtual processors that do not execute the `break` statement will continue to loop until they either execute a `break` statement, or the loop test becomes false.

The `break` fix up is similar to the `continue` fix up. First we create a poly temporary variable that reflects the break state, and an initialization statement that sets its value to one (meaning no break yet). The initialization statement is inserted immediately before the loop. The loop condition expression is modified to test the state of the break state variable before it tests the original loop condition, using the `&&` operator so that if the break state variable goes to zero, the loop condition will not be evaluated an extra time, which could give incorrect results if evaluating the original condition involves side effects. If no barrier synchronizations are needed between the `break` statement and the end of the loop, the `break` can be replaced with a compound statement that sets the break state variable to zero, and a `goto` statement that jumps to the end of the loop. However, if a barrier synchronization is required between the `break` statement and the end of a loop, the `goto` type fix up cannot be used (see `continue` for reasons why). Instead we enclose the remainder of the code in the loop in an `if` statement, whose condition is the break state variable. The `break` is replaced with an assignment statement that sets the break state variable to zero (false). Similar to the `continue` fix up, the `if` fix up can always be used, but we use the `goto` fix up whenever possible, because it results in more efficient code. Examples of both types of `break` fix ups are shown below:

Before `break` Fix up:

```
while (exp1)  {
  statement_block1;
  [sync]
  statement_block2;
  if (exp2) {
    break;
  }
  statement_block3;
}
```

After `goto` Fix up:

```
break_var = 1;
while (break_var && exp1) {
  statement_block1;
  [sync]
  statement_block2;
  if (exp2) {
    {
      break_var = 0;
      goto label;
    }
  }
  statement_block3;
label;
}
```

Before `break` Fix up:

```
while (exp1) {
  statement_block1;
  if (exp2) {
    break;
  }
  statement_block2;
  [sync]
  statement_block3;
}
```

After `if` Fix up:

```
break_var = 1;
while (break_var && exp1) {
  statement_block1;
  if (exp2) break_var = 0;
  if (break_var) {
    statement_block2;
    [sync]
    statement_block3;
  }
}
```

After all `break` and `continue` statements have been fixed up, the normal barrier synchronization transformations will result in correct code.

Return in Member Functions

We must transform `return` statements in member functions into equivalent code that does not use a `return` statement. If a `return` was left intact in a member function, when a virtual processor executed the `return`, it would cause the physical processor emulating it to return to the calling code. This would prevent that physical processor from emulating the rest of its virtual processors, or participating in any barrier synchronizations in the member function.

In the expression transformation discussion we explained the `return` member variable that we add to the domain type. The return variable is used to return the value of the member function. The `return` statements are replaced with an assignment to the return variable, if they are returning a value, and code to set their active variable to false. Remember, member function parse trees are built with a domain select with a `if` statement for the body with a poly active variable for the test. The surrounding active `if` statement is split so that the state of the active variable can be tested. We provide an example to help clarify the procedure.

Untransformed Code:

```
float foo_baz()
{
  [domain foo].{
    if (baz_active) {
      statement_block1;
      if (exp1) return exp2;
      statement_block2;
      return exp3;
    }
  }
}
```

Transformed Code:

```
float foo_baz()
{
  [domain foo].{
    if (baz_active) {
      statement_block1;
      if (exp1) {
      baz_return = exp2;
      baz_active = 0;
      }
    }
    if (baz_active) {
      statement_block2;
      baz_return = exp3;
      baz_active = 0;
    }
  }
}
```

The untransformed code shows how the member function was built during the scan and parse phase. Its body has been wrapped in a domain select and an `if` statement.

Domain Select Statement

Eventually all code splitting transformations will propagate up to the point where a domain select statement is reached. All that is required is to wrap the before parse tree with a domain select statement of an identical type, and insert it in the parse tree before the current domain select node. When they are unparsed, a barrier synchronization will be inserted between them.

Many of the statement and expression code transformations try to use existing temporaries. For example, if a transformed loop contains several `break` statements, only one break control variable will be created, and not one for every `break`. The same is true for `continue`. Splitting control structures and expressions do not create a new temporary if one is already in place, perhaps from a previous split. There are many places were we can perform this type of temporary reducing optimization.

8 Peephole Optimizer

When all the transformations required to split the domain selects and add barrier synchronization points have been completed, the compiler then performs a peephole optimization phase. The purpose of the peephole optimizations are primarily to clean up unnecessary code that was created either by the programmer, or more likely, by the code transformations to add barrier synchronizations. The optimizations performed are now outlined.

8.1 Expressions

Expressions whose values are not used and can be determined to contain no side effects, are removed. This can occur in the following instances:

1. If the left hand operand for a comma operator has no side effects, the comma operator can be replaced with the right hand operand.
2. Multiply nested parenthesis can be replaced with single set of parenthesis.
3. An expression in an expression statement with no side effects can be removed, leaving an empty expression statement.
4. Pre or post initializes in a for statement with no side effects can be removed.

8.2 Statements

Statements provide many potential opportunities for improvement. The transformations tend to generate messy code that is relatively easy for the compiler to clean up. The statement peephole optimizations performed are:

1. Empty expression statements are removed (expression peephole optimizations can produce these).
2. Multiply nested compound statements with no declarations can be reduced to a single compound statements. Barrier synchronization transformations produce this type of code.
3. Empty compound statements are removed.
4. Control structures with constant conditions can often be reduced.
5. Empty domain selects are removed.
6. `If` or `switch` statements with empty code bodies can be eliminated if the test expression has no side effects. If the test condition has side effects, the `if` or `switch` statement is replaced with an expression statement containing the test condition.
7. Nested `if`'s with identical test conditions can be merged if it can be determined that no side effects could alter the test condition.

The peephole optimizations work together to clean up the code. Many of these optimizations do not necessarily make for more efficient code, but the resulting code is easier for the programmer to read.

9 Unparse Phase

The last phase of the compiler is the actual code generation phase. The unparse phase traverses the entire parse tree emitting C code. Most of the parse tree will be standard C, and can be emitted "as is". There may be several types of nodes in the parse tree that will require special attention to unparse. These nodes are the function declaration for `main` and all references to it, member functions declarations and invocations, communication macro initializations, domain declarations, min/max operators, reductions, domain selects, and array assignment.

9.1 Main

When unparsing `main`, we rename its declaration and all references as `__main`. This is because our start-up code that is linked in provides a `main` that allocates the desired number of processors for use in emulating the domain selects. The start-up main then calls `__main`. We chose this approach instead of writing our own crt0.o start-up code, because it was easier to develop and made porting simpler.

9.2 Member Functions

Previously we have explained how member functions are called from parallel code. The method we outlined will not work when invoking a member function from sequential code for two reasons. Member functions provide their own virtual processor emulation, but they expect to be called in multi mode. Our run-time model keeps the system in single mode in sequential code. The other problem is all the active variables must be initialized to true, and the arguments initialized. The return value from the member function must also be correctly handled.

Our solution to these problems has been to provide a helper function for each member function. All calls to member functions from sequential code are unparsed to instead call the helper function. The helper function performs that tasks of initializing the active variables and all argument variables. It then invokes the real member function from multi mode and correctly returns the correct return value. As a short example consider:

Untransformed Sequential Code:

```
x = foo::baz(j, 3)/k;
```

Transformed Code and Helper Function:

```
float sequential_foo_baz(i,j)
int i, j;
{
  [domain foo].{
    baz_arg_i = i;
    baz_arg_j = j;
    baz_active = 1;
  }
  /* still in multi mode */
  foo_baz();
  m_single();
  return bar[0].baz_return;
}

 ...

x = sequential_foo_baz(j, 3)/k;
```

9.3 Domain Declarations

Domain declarations are unparsed as structures. The member function prototypes are removed for this purpose, since structure declarations cannot have member function references. The domain declaration

```
domain foo {
  int i, j, k;
  float x[20];
  float baz();
} bar[200][200];
```

Is unparsed as

```
struct foo {
  int i, j, k;
  float x[20];
};
float foo_baz();
struct foo bar[200][200];
```

9.4 MIN/MAX Operators

Since C has no built-in min and max operators, we unparse the min and max operators as calls to special min and max functions we provide. Functions are used instead of macros, because Dataparallel C specifies that the arguments to min and max will be evaluated only once. Min and max functions must be provided for types double, signed integer, and unsigned integer. The expression

```
x = y <? z;
```

is expanded into

```
x = __double_min(y, z);
```

Many modern C compilers, such as GNU C and the Sequent C compilers, could expand these functions in-line, giving the same performance as macros.

Dataparallel C also includes the assignment forms of min and max. Functions are provided for these operators. The expression

```
x <?= y;
```

would be expanded into

```
__binary_double_min(&x, y);
```

Here the address of x is passed as an argument, because x may need to be modified by __binary_double_min. The function returns the value of the minimum value, incase this value is subsequently used, as in

```
z = x <?= y;
```

9.5 Array Assignment

Since arrays and sub-arrays (in the case of a partial array assignment) are laid out in contiguous memory, we can replace array assignments with calls to the C bcopy library function. During data flow array assignment nodes are marked with the number of elements being assigned. This information, along the address of the arrays being assigned, is used as arguments to the bcopy routine. For example, given the array declarations

```
float x[100][10], y[100][10];
```

the code

```
x = y;
```

will be unparsed as

```
(bcopy(y, x, 1000 * sizeof(float)), x);
```

The comma operator is used so the expression will evaluate to a pointer to array x, incase this value is used (perhaps in another array assignment, as in z = x = y).

9.6 Domain Selects

Domain select nodes contain information specifying the domain type, the number of virtual processors, the parallel/sequential mode the domain select is entering and exiting in, and if any special entry or exit code must be generated for a reduction or deterministic selection assignment operator. This information is collected during the parse and the code transformation phases.

Entry Modes

There are several possible entry modes for a domain select. Initially the domain select will be entered in single mode; only one processor will be active, and all the others must be activated before emulating the virtual processors. The Sequent parallel programming library provides the `m_multi` function to perform just that purpose. All physical processors that were previously allocated with the `m_fork` function call will become active on a call to `m_multi`. Unfortunately `m_multi` has a serious flaw; after executing a `m_multi`, the registers for the slave processors will not match the registers for the master processors (processor zero). Also, C compilers allocate local storage on the stack for all block scoped variables declared in a procedure at the entry point for the procedure, regardless of where the block scoped variables are actually declared in that procedure. This can cause a problem with our run-time model, because the C compiler being used to compile our translated program has no knowledge that it is compiling a parallel program, and may make incorrect assumptions about the contents of registers before and after invoking `m_multi`. We emit code that guarantees the states of all slave processors will match the master processor. To accomplish this we have two assembly language macros that are expanded in-line in the C code. The first is `__WRITEREGS()`. This macro is emitted immediately before the call to `m_multi`, so it will only be executed by the master processor. It saves the contents of all its registers in a set of shared variables. The second assembly language macro is `__READREGS()`. We emit a reference to this macro immediately after the `m_multi` function call, which will be parallel code. Each physical processor can use the values of the saved stack pointer and frame pointer to compute how large a local stack it will need. Once the stacks have been setup, they load their general purpose registers with the saved values. (Setting up the stacks does not copy the values from the front-end stack to the slave processor stacks. This is because all mono variables referenced in parallel code will have already been moved to global shared memory. We just need to make sure there is storage for any locals used in the parallel code.)

If the domain select is being entered already in multi mode, the registers and stack will already be correct. This occurs after a barrier synchronization split in a domain select.

Exit Modes

There are three possible exit modes from a domain select. If the domain is exiting to perform a barrier synchronization, and it will immediately be entering another domain select, then all that is emitted is a call to `m_sync`, which will perform a barrier synchronization but leave all processors in multi mode.

If the domain select is exiting, and it is immediately followed by sequential code, then it must emit code to shut down the slave processors. This occurs when the domain just exited was the tail of an original domain select in the Dataparallel C source code, and for some reduction operators. The code emitted is a call to the `m_single` function.

The while loop transformation described earlier also required the ability to exit a domain select, stay in multi mode, but perform no barrier synchronization. This requires that no extra code be emitted.

9.7 Virtual Processor Emulation

The virtual processors themselves are emulated with a for loop, where each physical processor iterates over its share of virtual processors using the variable `__this` as an iterator. Variable `__this` is declared to be a local pointer to the current domain type in a locally scoped block. There are two different types of virtual processor emulation loops we can emit. One provides contiguous virtual processor emulation, and the other provides interleaved virtual processor emulation. In contiguous virtual processor emulation, each physical processor emulates a group of virtual processors representing a contiguous block of the active domain instances. In interleaved virtual processor emulation, a physical processor starts with the n'th virtual processor and emulates the k'th virtual processor, where n is the unique id of the physical processor (from 0 to numprocs-1), and k is the number of physical processors emulating virtual processors. For well balanced applications contiguous tends to give slightly better results, because of a better cache hit rate. For poorly balanced applications interleaved gives better results, because it does a better job of load balancing. We provide examples of both interleaved and contiguous virtual processor emulation.

For the domain declaration

```
domain foo.{
  int i, j;
} bar[10][20];
```

and the domain select

```
[domain foo].{
  statement_body;
}
```

The interleaved emulation would be

```
__WRITEREGS();
m_multi();
__READREGS();
{
  struct foo *__this, *__thisend;

  __this = &bar[0][_myid];
  __thisend = &bar[0][200];

  for (; __this < __thisend; __this += __numprocs) {
    statement_body;
  }
}
m_single();
```

and the contiguous emulation would be

```
__WRITEREGS();
m_multi();
__READREGS();
{
  struct foo *__this, *__thisend;
  int __size;

  __size = (((__size = 200 / __numprocs) * __numprocs == 200) ?
          __size :
          __size+1;

  __this = &bar[0][__size*_myid];
  __thisend = ((__this + __size > &bar[0][200]) ?
             &bar[0][200] :
             __this + __size);

  for (; __this < __thisend; __this++) {
    statement_body;
  }
}
m_single();
```

Note that although the domain is a 10 by 20 two dimensional array, virtual processors are only emulated by a single loop. We take advantage of C laying the array out in contiguous memory and treat it as a single dimensional array with 200 elements. This saves the overhead of nesting loops to perform virtual processor emulation.

For both the interleaved and contiguous methods of virtual processor emulation, each physical processor uses its physical id to compute where to start, and the number of physical processor active to compute what virtual processors to emulate.

9.8 Communication Macro Initializations

Because communication macros are expanded into special member pointer references, these pointers must be initialized before they can be used. When main is unparsed, each domain type is checked to see which, if any, communication macros were used for that domain type. Code is then emitted to initialize the member pointers for the used macros. The macros are initialized in parallel, with each processor initializing its share of the virtual processors, much as if it were being done in a domain select. As an example of initializing `north()` and `west()` for a 5 by 10 domain, consider

```
int __main()
{
  if (!__communication_macros_inited) {
    __communication_macros_inited = 1;

    __WRITEREGS();
    m_multi();
    __READREGS();
    {
      struct foo *__this, *__thisend;

      __this = &bar[0][_myid];
      __thisend = &bar[0][20000];

      for (; __this < __thisend; __this += __numprocs) {
      __this->__north = ((__this > &bar[0][199]) ?
                          __this - 200 :
                          __this + 19800);

      __this->__west = (((__this - &bar[0][0]) % 200) ?
                         __this - 1 :
                         __this + 199);
      }
    }
    m_single();

    ...

}
```

The complexity of these expressions is necessary to correctly handle the toroidal wrap for the boarders.

Notice that the initialization of the pointers is executed conditionally. This is to prevent the initialization from being done repeatedly if main is called recursively.

9.9 Reductions and Deterministic Selective Assignments

Reductions require some extra code to be emitted outside the domain select. For example, to perform a min reduction, a local and global minimum variable must first be initialized to a maximum value. Each physical processor computes the local reduction of all the virtual processors it is emulating in the domain select code. After exiting the domain select, code must be emitted so each physical processor uses the local reduction it has performed to compute the global reduction. To perform a deterministic selective assignment, the variable that contains the value of the `__this` pointer that performed the current assignment must be initialized to a maximum value. What follows is an example of both a min reduction and a selection assignment.

Untransformed Code:

```
[domain foo].{
  poly_i = <?= poly_j;
  mono_k = poly_l;
}
```

Unparsed Transformed Code:

```
tmp1 = _MAXSIGNEDINT_;
{
  int tmp2 = _MAXSIGNEDINT_;
  int tmp3;

  VP_EMULATION_LOOP {
    if (poly_j < tmp2) tmp2 = poly_j;
  }
  if (tmp2 < tmp1) {
    m_lock();
    if (tmp2 < tmp1) {
      tmp1 = tmp2;
    }
    m_unlock();
  }
}
tmp4 = _MAXPTR_;
{
  VP_EMULATION_LOOP {
    if (this < tmp4) {
      m_lock();
      if (this < tmp4) {
      tmp4 = this;
      mono_k = poly_l;
      }
      m_unlock();
    }
  }
}
```

The above code shows how the local reduction is performed first for the min reduction. Nesting the if statements, and only locking the inner if is used to reduce the number of locks performed.

10 Compiler Performance

In [8, 11, 21, 25] the performance of our compiler is discussed for a number of benchmark programs. We encourage interested readers to investigate these references further. We emphasize that not all the benchmarks are from the "standard set" of toy problems, but we include some applications from other disciplines, as well as an exhaustive search problem. The applications benchmarked are:

Matrix Multiplication – Classical $\Theta(n^3)$ algorithm for matrix multiplication.

Gaussian Elimination – Gaussian Elimination with partial pivoting and back substitution.

Gauss Jordan – Solving system of linear equations using the Gauss Jordan method. Also performs partial pivoting.

Numerical Integration – Calculating the value of π by integrating $4/(1+x^2)$ between 0 and 1.

Transitive Closure – Uses Warshall's transitive closure algorithm to find the transitive closure of an adjacency matrix.

Computation of Prime Numbers – Uses the Sieve of Erotosthenes to find prime numbers.

Computation of Relative Prime Numbers – Uses Euclid's gcd algorithm to compute relatively prime numbers.

The Triangle Puzzle – A search problem suggested by Foster and Taylor [5].

Sharks World – An ICASE watery world simulation with fish and sharks. See [17] for more information.

Wa-Tor – Another watery world fish and shark simulation. The simulation was first described in [4].

Shallow – An atmosphere model developed at the National Center for Atmospheric Research in Boulder Colorado.

Ocean – A layered ocean circulation simulation based on the model in [3].

Two tables showing the execution times and absolute speedups for the Sequent Symmetry, based on the best sequential programs we were able to develop, are shown below. The speedups for the Sequent Balance are similar. We are enthusiastic about the speedups these applications have been able to achieve, and believe this gives strong credibility to our research. We feel we have been able to successfully meet our goals of translating a SIMD Dataparallel C program into an efficient SPMD C program.

Program	Size	Sequential Execution (sec)	Execution Time of Dataparallel C Programs (sec)						
			Number of Processors						
			1	4	8	12	16	20	24
pi	100,000	3.20	3.78	0.95	0.48	0.32	0.24	0.19	0.16
pi	200,000	6.42	7.58	1.89	0.95	0.64	0.47	0.39	0.32
pi	400,000	12.80	15.20	3.79	1.89	1.27	0.95	0.76	0.63
relprime	128	0.27	0.30	0.08	0.04	0.03	0.02	0.01	0.02
relprime	256	1.36	1.47	0.39	0.19	0.13	0.10	0.08	0.07
relprime	512	6.66	6.98	1.84	0.94	0.64	0.48	0.38	0.33
matrix	64	3.35	4.06	1.02	0.51	0.38	0.26	0.26	0.19
matrix	128	33.70	33.10	8.30	4.15	2.86	2.09	1.83	1.57
matrix	256	259	269	66.70	33.50	26.10	16.80	13.70	11.60
warshall	64	0.90	1.21	0.31	0.16	0.12	0.09	0.08	0.07
warshall	128	7.58	9.27	2.35	1.19	0.82	0.61	0.54	0.47
warshall	256	48.30	78.50	18.60	9.31	6.30	4.64	3.78	3.22
gauss	64	1.68	1.67	0.53	0.33	0.30	0.28	0.22	0.35
gauss	128	12.80	12.20	3.34	1.84	1.48	1.23	1.23	1.23
gauss	256	103	93.20	24.80	13.30	9.94	7.50	6.48	6.16
g-jordan	64	-	2.23	0.60	0.33	0.27	0.23	0.25	0.24
g-jordan	128	-	17.10	4.23	2.21	1.58	1.22	1.12	1.05
g-jordan	256	-	135	33.70	17.00	11.70	8.68	7.30	6.40
sieve	1,200,000	4.30	8.86	2.18	1.07	0.77	0.65	0.56	0.42
sieve	2,400,000	8.84	17.50	4.38	2.22	1.57	1.27	1.07	0.84
sieve	4,800,000	18.10	34.90	8.76	4.47	3.25	2.70	2.35	1.69
triangle	all solns	73.30	73.30	19.90	12.10	8.22	7.19	5.76	4.98
shallow	64x64	2681	2082	509	255	172	127	102	85.8
ocean	640	2708	2724	626	308	212	163	139	128
sharks	2048/2,000	448	417	108	55.5	37.2	28.5	22.9	19.1
sharks	4096/2,000	1793	1650	442	233	163	114	90.5	84.4
sharks	2048/100,000	1309	1584	402	202	138	105	84.1	70.2
sharks	4096/100,000	5841	5736	1602	810	552	416	343	284

Program	Size	Absolute Speedup of Dataparallel C Programs						
		Number of Processors						
		1	4	8	12	16	20	24
pi	100,000	0.85	3.37	6.67	10.00	13.33	16.84	20.00
pi	200,000	0.85	3.40	6.76	10.03	13.66	16.46	20.06
pi	400,000	0.84	3.38	6.77	10.08	13.47	16.84	20.32
relprime	128	0.90	3.38	6.75	9.00	13.50	27.00	13.50
relprime	256	0.93	3.49	7.16	10.46	13.60	17.00	19.43
relprime	512	0.95	3.62	7.09	10.41	13.88	17.53	20.18
matrix	64	0.83	3.28	6.57	8.82	12.88	12.88	17.63
matrix	128	1.02	4.06	8.12	11.78	16.12	18.42	21.46
matrix	256	0.96	3.88	7.73	9.92	15.42	18.91	22.33
warshall	64	0.74	2.90	5.63	7.50	10.00	11.25	12.86
warshall	128	0.82	3.23	6.37	9.24	12.43	14.04	16.13
warshall	256	0.62	2.60	5.19	7.67	10.41	12.78	15.00
gauss	64	1.01	3.17	5.09	5.60	6.00	7.64	4.80
gauss	128	1.05	3.83	6.96	8.65	10.41	10.41	10.41
gauss	256	1.11	4.15	7.74	10.36	13.73	15.90	16.72
g-jordan	64	0.75	2.80	5.09	6.22	7.30	6.72	7.00
g-jordan	128	0.75	3.03	5.79	8.10	10.49	11.43	12.19
g-jordan	256	0.76	3.06	6.06	8.80	11.87	14.11	16.09
sieve	1,200,000	0.49	1.97	4.02	5.58	6.62	7.68	10.24
sieve	2,400,000	0.51	2.02	3.98	5.63	6.96	8.26	10.52
sieve	4,800,000	0.52	2.07	4.05	5.57	6.70	7.70	10.71
triangle	all solns	1.00	3.68	6.06	8.92	10.19	12.73	14.72
shallow	64x64	1.29	5.27	10.51	15.59	21.11	26.28	31.25
ocean	640	0.99	4.33	8.79	12.77	16.61	19.48	21.16
sharks	2048/2,000	1.07	4.15	8.07	12.04	15.72	19.56	23.46
sharks	4096/2,000	1.09	4.06	7.70	11.00	15.73	19.81	21.24
sharks	2048/100,000	0.83	3.26	6.48	9.49	12.47	15.56	18.65
sharks	4096/100,000	1.02	3.65	7.21	10.58	14.04	17.03	20.57

11 Future Work

There are a number of ways in which Dataparallel C and our compiler might be improved. These possible improvements fall into two categories, language improvements and compiler improvements.

11.1 Language Improvements

As the language currently stands, there are issues that it addresses poorly or not at all. Some of these issues are:

Multiple SIMD modules – An application may consist of different SIMD type modules that could be executed either in a MIMD or pipelined fashion. Currently there is no mechanism to overlap execution of SIMD modules.

Dynamic Parallelism – The language allows no dynamic creation or destruction of domain instances. This would be a convenient addition to the language, and would benefit applications where the size of the problem is not known at compile time.

Nested Parallelism – The language only supports flat parallelism. It may be useful in some instances to allow domain selects inside parallel code.

Generic Parallel Library Routines – Code reusability is an important issue in software development. The ability to write generic parallel functions that could be invoked from any parallel code would allow programers to reuse routines and develop libraries of parallel routines.

Better Communication Scheme – The current communication scheme for the language relies heavily on the use of communication macros for efficient execution, because the language does not provide a good mechanism for specifying general communications (we do not feel that using pointer arithmetic and `this` provide a good communication scheme).

11.2 Compiler Improvements

It should be understood that the compiler we have developed is not a toy. We have written a robust compiler that implements the full Dataparallel C language in an efficient way. However, there are still ways the compiler could be improved. Most, but not all the improvements deal with optimizations.

11.2.1 Data flow Limitations

The primary purpose of data flow is to determine where barrier synchronizations must be enforced. When data flow cannot determine if there is a dependency, it assumes there is one, possible introducing unneeded barrier synchronizations into the program.

Our data flow analysis is not perfect, and could be improved upon in several ways. Currently, data flow for pointers is only computed for pointers to domain types. This can lead to potentially incorrect code generation for pointers that point to members of domains, because in these instances we do not err on the conservative side. Also, functions invoked from inside parallel code should be considered uses and defs of all global data items, since we do not perform intra-procedural data flow. This is not a problem for member functions, because we automatically insert a barrier synchronization before and after the function call, but this is not the case for sequential functions called from within parallel code. Also, such techniques as maintaining possible values for variables such as pointers and variables used as array subscripts, could help reduce the number of unnecessary barrier synchronizations. Our experience has led us to believe these are not serious limitations as long as the programmer is aware of them.

Data flow is also a rather expensive computation, and can require significant quantities of time and memory. Improvements could be made the reduce these requirements, such as those outlined in [1]. The data flow routines were implemented with the objective of making the implementation task as easy as possible for the implementor, often with little concern for the computational complexity. As a result the compiler can be slow, especially when the program contains nested loops.

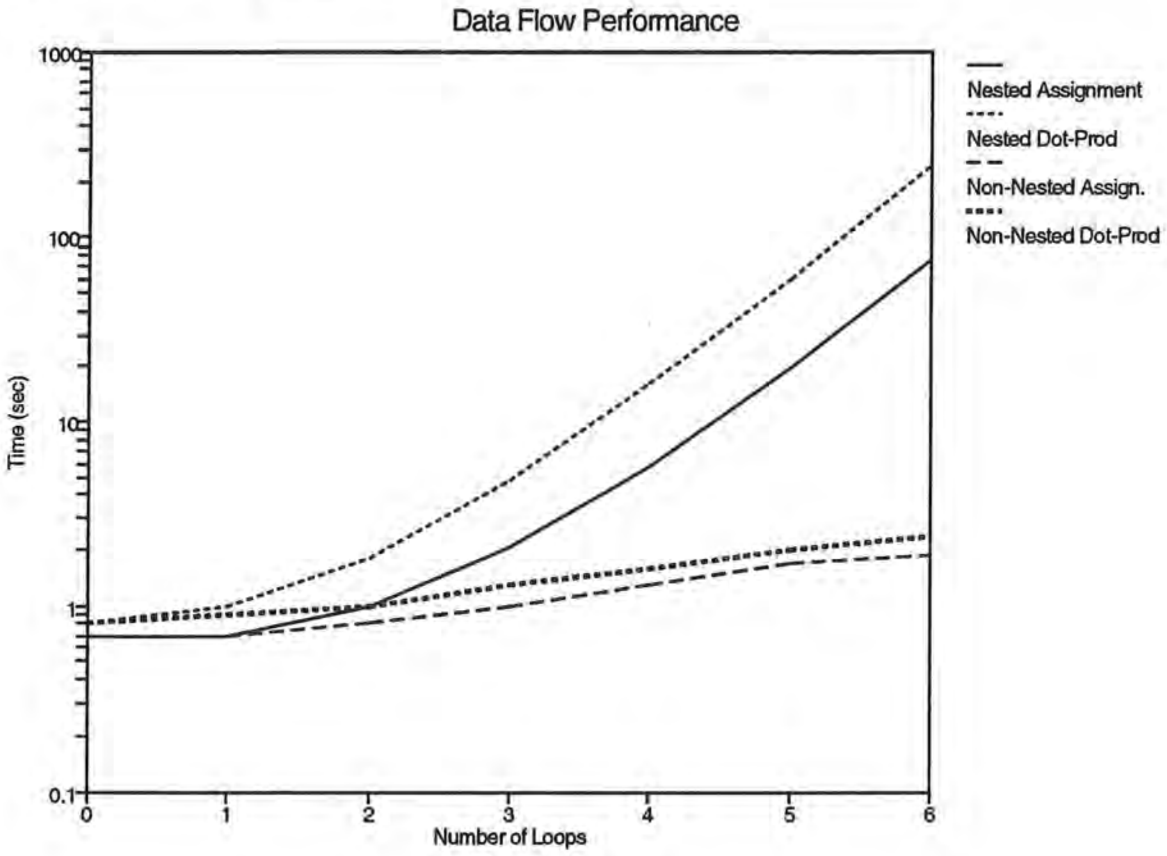
The graph below contains compile times for four different programs to demonstrate the time costs of performing data flow on nested loops. Two of the programs contained nested loops, while the other two contained non-nested loops. The number of loops was varied from zero to six. Each program also contained a single expression statement,

```
poly = this - &bar[0];
```

for half of the programs, and

$poly_1[i]$ += $poly_2[i]$ * $poly_3[i]$;

for the other half. All programs were under 20 lines long.



The data above was collected for very simple programs. It shows that the computational complexity not only increases exponentially with the number of nested loops, but increases also with the number of variable uses and defs in the code, as well as the complexity of the expressions they are used in. It is quite possible to have relatively short Dataparallel C programs (a few hundred lines) that require half an hour to an hour to compile on the Sequent Symmetry.

This should not be considered a fault of Dataparallel C, but of our implementation of data flow. In practice, most programs seem to compile in several minutes.

11.2.2 Temporary Variable Reuse

Whenever the compiler needs a temporary variable, a new variable is introduced into the current domain type. This can cause a large number of temporaries to be introduced over the compilation of an entire program. Often, when a temp is needed, an unused temporary will already be available in the domain. Reusing variables could significantly reduce the memory requirements of the program. This would require maintaining live/dead information for the temporaries, but we do not feel this would be difficult to implement.

Another possible solution would be to not add temporaries to the domain type, but instead declare local arrays of temporaries. They would then be allocated on the stacks of the physical processors. The difficulty in this approach is that the current virtual processor emulation scheme iterates using `this`, which is also used to access data items. Since `this` is a pointer, pointing to the domain instance array, it could not be directly used to access these temporaries, and a more costly mechanism for accessing them would be needed.

11.2.3 Intra-procedural Data Flow and In-line Expansion of Member Functions

The compiler only performs inter-procedural data flow analysis. As a consequence of this, when generating code for a member function invocation from inside parallel code, the compiler does not know if there are any dependencies that must be preserved. The only safe course of action is to execute a barrier synchronization immediately before and after the member function call, as well as any barrier synchronizations needed in the body of the member function. As an example, a member function invocation currently generates the following code:

```
[domain foo].{
  ...
  foo::bar();
  ...
}
```

becomes:

```
[domain foo].{
  ...
}
m_sync();
foo::bar();
m_sync();
[domain foo].{
  ...
}
```

If intra-procedural data flow analysis was performed, better code could be generated in several cases. The first case is if there are no dependencies in the parallel code before the function call and the member function, or no dependencies between the member function and the code following the call. If this is the case, one or both of the barrier synchronizations may be omitted. This would give the code:

```
[domain foo].{
   ...
}
foo::bar();
[domain foo].{
   ...
}
```

Note that the function call still must be brought out of the domain select into serial code, because there may still be barrier synchronizations inside the member function. However, if there are not, and we would know this from the intra-procedural data flow analysis, then we could produce the original code:

```
[domain foo].{
   ...
   foo::bar();
   ...
}
```

This could be a huge savings in several ways. First, if the function is being invoked conditionally, virtual processors who do not need to invoke it will not, thus saving time. Secondly, arguments could be passed to the member function in a regular manner, and no special mechanism for accessing the return value would be needed. This would save both time and memory space, because there would be no need to created entries for the arguments in the domain declarations as well as no need for the active vector currently needed inside the member functions.

In-line expansion of member functions could also improve code by eliminating the call overhead, possibly increasing grain size, reducing the number of barrier synchronizations, and allowing a higher degree of optimization by the C compiler used to compile the resulting C program. In the presence of in-line expansion of member functions, many of the benefits of intra-procedural data flow could be realized without intra-procedural data flow.

11.2.4 Reduce the Number of Barriers by Moving Uses Up and Defs Down

If uses of variables were moved up, and defs of variables moved down, the number of barrier synchronizations could be reduced in some instances. For example, given the code fragment:

```
i = exp1;
j = exp2;
k = exp3;

west()->i = i;
west()->j = j;
west()->k = k;
```

This will currently generate the following code with 3 syncs:

```
i = exp_1;
j = exp_2;
k = exp_3;

tmp_i = i;
[sync]
west()->i = tmp_i;

tmp_j = j;
[sync]
west()->j = tmp_j;

tmp_k = k;
[sync]
west()->k = tmp_k;
```

This code is poor not only because of the relatively high number of barriers, but also because of the small grain size of the resulting virtual processor emulation loops.

If instead, we moved the uses of `i`, `j`, and `k` up as far as we could, the following code could be generated:

```
i = tmp_i = exp_1;
j = tmp_j = exp_2;
k = tmp_k = exp_3;

[sync]

west()->i = tmp_i;
west()->j = tmp_j;
west()->k = tmp_k;
```

The number of barrier synchronizations has now been reduced to one, and the virtual processor emulation grain size has been improved.

This is an optimization that the programer could do at the source level, but it is probably better done by the compiler (make the programmer's job simpler, and less error prone. Also ensure that we catch all cases). To move the uses up, the compiler needs to know for every use of a variable, where all the previous live defs of that variable are. There may be more than one in the case of conditional assignments. A use may not be moved before any of its defs.

Similar savings could be made by moving defs down as far as possible. To do this the compiler would need to know for each def, where the next uses of that variable are so that it could be moved just previous to them.

Currently the compiler performs all the data flow necessary for these tasks, so we would not think that these optimizations would be too difficult to implement.

11.2.5 Unroll virtual processor emulation loops with small code bodies

Sometimes after adding all the necessary barrier synchronizations, the resulting bodies of the domain selects are quite small, possible a single expression. When this is the case, the overhead for emulating the domain select may be higher than the work being done by the domain select. For example, consider the domain select

```
[domain foo].{
  i = 2;
}
```

Every virtual processor must perform a single simple assignment. The work to emulate each virtual processor is an addition, a compare, and jump, which is more expensive than the assignment.

The loop could be unrolled to make the loop overhead virtually as low as we would like. However, we would not want to unroll a loop so far that it would no longer fit in the system's instruction cache.

11.2.6 Allow multiple reduce type ops at the end of selects

Every reduction or deterministic selection assignment currently generates a barrier synchronization. This is so the code can be generated outside the virtual processor emulation loop to initialize certain variables before the emulation loop or to perform the final global reduction at the end of the loop. It would be possible to allow multiple reduction initializations or global reductions to be performed for the same virtual processor emulation loop. This could reduce the number of barrier synchronizations and increase the grain size of the parallel code.

12 Summary and Conclusions

There are a number of areas in which our research makes an original contribution. In creating Dataparallel C, we modified the semantics of C* slightly to "make more sense". Our compiler targets code to tightly coupled multiprocessors instead of SIMD architectures or distributed memory multicomputers as in [20, 22, 23]. We use data flow techniques to perform optimizations and generate efficient code in the translation of SIMD programs to SPMD programs with identical semantics. We modified the *SelectSyncs* algorithm to use data dependency spans in selecting barrier synchronization points. The code transformations used in this translation process needed to be developed. A suitable and efficient run-time model was developed to support the translation scheme. We developed and implemented several optimizations to improved performance of the translated codes. And finally, we implemented a number of algorithms in Dataparallel C and evaluated our approach on both the Sequent Symmetry S81 and the Sequent Balance 21000.

After using our Dataparallel C compiler for some time now, we are encouraged by the results of our research. Writing parallel programs for the Sequent computers using the tools provided by Sequent is an arduous task for all but very simple applications. We feel using our Dataparallel C compiler makes parallel programming as easy as sequential programming, once the parallel algorithm for the problem is understood. Data-parallel programming feels to us much like programming in a regular sequential language due to the single locus of control, and we have found the code-debug cycle is much the same.

Not only do we feel Dataparallel C is much easier to write many parallel programs in than other conventional methods for programming MIMD machines, we have shown it is possible to efficiently compile data-parallel languages for MIMD computers, yielding programs whose execution times are very competitive to the best hand coded programs.

We believe now more than ever that data-parallel programming on MIMD machines gives the programmer the best of both worlds, ease of programming and the ability to code efficient programs. There are some types of programming that we feel are more easily performed using Dataparallel C than conventional sequential C, such as the implementation of cellular automata.

We hope that this research will help convince others that the data-parallel programming paradigm on MIMD architectures is a very attractive option for many applications, and may in many cases may be the best option currently available.

Our Dataparallel C compiler consists of about 35,000 lines of C code. It is a full implementation of the Dataparallel C language. The compiler has already been used for several parallel programming courses at Oregon State University, as well as other programming projects, and seems to be fairly robust. Versions are available for both the Sequent Symmetry and Balance multicomputers as well as a single processor UNIX workstation version. We are making the compiler sources available to all interest parties.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools,* Reading, Massachusetts: Addison-Wesley, 1986.

[2] W. M. Ching, "Automatic parallelization of APL-style programs," In *Proc. of APL'90 Conf,* pp. 76-80, 1990.

[3] R. Courant and D. Hilbert, *Methods of Mathematical Physics,* vol. 2, Wiley Interscience, 1962.

[4] A. K. Dewdney, "Computer recreations," Scientific American, pp. 14-22, Dec. 1984.

[5] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming.* Englewood Cliffs, N. J.: Prentice Hall, 1990.

[6] G. C. Fox, "What have we learnt from using real parallel machines to solve real problems?" In *Proc. Third Conf. Hypercube Concurrent Computers and Applications,* pp. 897-955, ACM, New York, 1988.

[7] L. H. Hamel, P. J. Hatcher, and M. J. Quinn, "An optimizing compiler for a hypercube multicomputer," In *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines,* Elsevier, 1991.

[8] P. J. Hatcher, M. J. Quinn, R. J. Anderson, A. J. Lapadua, B. K. Seevers, and A. F. Bennett, "Architecture-independent scientific programming in Dataparallel C: Three case studies," In *Supercomputing '91.*

[9] P. J. Hatcher, M. J. Quinn, A. J. Lapadua, R. J. Anderson, and R. R. Jones, "Dataparallel C: A SIMD programming language for multicomputers," *Proceedings of the Sixth Distributed Memory Computer Conference,* To appear.

[10] P. J. Hatcher, M. J. Quinn, A. J. Lapadua, R. R. Jones, and R. J. Anderson, "A production-quality C* compiler for a hypercube multicomputer," In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* Apr. 1991.

[11] P. J. Hatcher, M. J. Quinn, A. J. Lapadua, B. K. Seevers, R. J. Anderson, and R. R. Jones, "Data-parallel programming on MIMD computers," *IEEE Transactions on Parallel and Distributed Systems,* July 1991.

[12] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Communications of the ACM,* vol. 29, pp. 1170-1183, Dec. 1986.

[13] W. D. Hillis and G. L. Steele, Jr., "Update to data parallel algorithms," *Communications of the ACM,* vol. 30, pp. 78, Jan. 1987.

[14] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "Supporting shared data structures on distributed memory architectures," In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pp. 177-186, 1990.

[15] K. Li and H. Schwetman, "Vector C: A vector processing language," In *Journal of Parallel and Distributed computing*, vol. 2, pp. 132-169, 1985.

[16] D. May, "OCCAM," ACM SIGPLAN Notices, vol. 18, pp. 69 – 79, April 1983.

[17] D. M. Nicol and S. E. Riffe. "A 'conservative' approach to parallelizing the Sharks World Simulation," Technical Report 90-67, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA., 1990.

[18] Osterhaug, Anita (Ed.), *Guide to Parallel Programming On Sequent Computer Systems,* Englewood Cliffs, N.J.: Prentice Hall, 1989.

[19] D. A. Padua, and M.J. Wolfe. "Advanced compiler optimizations for supercomputers," Communications of the ACM Vol. 29, pp. 1184-1201, December 1986.

[20] M. J. Quinn, and P. J. Hatcher. "Compiling SIMD programs for MIMD architectures," In *Proceedings of the 1990 International Conference on Computer Languages,* pp. 291-296, IEEE Computer Society Press, 1990.

[21] M. J. Quinn and P. J. Hatcher, *Data-Parallel Programming on MIMD Computers,* Cambridge, Massachusetts: MIT Press, To Appear.

[22] M. J. Quinn and P. J. Hatcher, "Data-parallel programming on multicomputers," *IEEE Software,* vol. 7, pp. 69-76, Sept. 1990.

[23] M. J. Quinn, P. J. Hatcher, and K. C. Jourdenais, "Compiling C* programs for a hypercube multicomputer," *SIGPLAN Notices,* 23, 9 (September), pp. 57-65. Proceedings of the ACM/SIGPLAN PPEALS 1988—Parallel Programming: Experience with Applications, Languages, and Systems, 1988.

[24] M. J. Quinn, P. J. Hatcher, and B. K. Seevers, "Implementing a data parallel language on a tightly coupled multiprocessor," In *Advances in Languages and Compilers for Parallel Processing,* pp. 385-401. Pitman.

[25] M. J. Quinn, P. J. Hatcher, and B. K. Seevers, "Implementing a time-driven simulation on a MIMD computer using a SIMD language," *International Journal of Computer Simulation,* To appear.

[26] A. P. Reeves, "Parallel Pascal: An extended Pascal for parallel computers," In *Journal of Parallel and Distributed Computing*, vol. 1, pp. 64-80, 1984.

[27] J. R. Rose and G. L. Steele, Jr., "C*: An extended C language for data parallel programming," Tech. Rep. PL 87-5, Thinking Machines Corporation, 1987.

[28] Thinking Machines Corporation, Cambridge, MA., *C* Programming Guide,* Version 6.0 Beta. Aug. 1990.

[29] Thinking Machines Corporation, "Connection Machine programming in *Lisp," Connection Machine documentation set.

[30] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers,* Pitman Publishing, London England, 1989.