1-1-1994

ViC*: A Preprocessor for Virtual-Memory C*

Thomas H. Cormen
Dartmouth College

Alex Colvin
Dartmouth College

Dartmouth College Computer Science
Technical Report PCS-TR94-243

ViC*: A Preprocessor for Virtual-Memory C*

Thomas H. Cormen*
Alex Colvin†
Dartmouth College
Department of Computer Science

Abstract

This paper describes the functionality of ViC*, a compiler-like preprocessor for out-of-core C*. The input to ViC* is a C* program but with certain shapes declared `outofcore`, which means that all parallel variables of these shapes reside on disk. The output is a standard C* program with the appropriate I/O and library calls added for efficient access to out-of-core parallel variables.

1 Introduction

Although parallel computers were originally designed with processing speed in mind, they have proven equally valuable for their ability to solve problems with very large data requirements. Indeed, parallel computers have opened up a new range of possibilities for scientific computing. Perhaps the best known examples are "Grand Challenge" problems such as those from environmental modeling, coupled field computations, and computational fluid dynamics.

As the capacity of parallel computers has increased, however, so have the appetites of users. Throughout the history of electronic computing, no matter how big and fast the top machines have been, there have always been applications that needed them to be bigger and faster, and it remains true today.

Over thirty years ago, computer architects devised virtual memory to solve this problem for sequential machines [Den70]. We see two approaches in today's parallel machines:

- Have no built-in support for virtual memory. Without built-in virtual-memory support, applications whose memory requirements exceed the available memory typically keep their data on a disk system and perform explicit disk accesses. The explicit I/O calls consequently add to the programmer's task, thus increasing the time spent in software development. The programmer spends time that could otherwise be spent on application-specific code instead worrying about disk I/O. Moreover, because I/O is often tricky to program, correct code is more difficult to develop and maintain.

*Supported in part by funds from Dartmouth College and in part by the National Science Foundation under Grant CCR-9308667. Author's email address: `thc@cs.dartmouth.edu`.

†Author's email address: `Alex.Colvin@dartmouth.edu`.

- Run traditional sequential virtual memory on the individual nodes, which requires a MIMD architecture. This approach frees the programmer from coding explicit I/O calls, but because it fails to take advantage of aggregate data-parallel operations, it also yields suboptimal performance. There have been significant technical advances recently on how to carefully plan disk accesses for common data-parallel operations and algorithms [CGG+94, Cor92, Cor93, CSW94, GTVV93, Kot94, NV91, NV92, VS94, WGWR93]. The performance improvements gained by using these methods can be tremendous, and their impacts increase with the problem size. They require a degree of coordination among the processors and disks that unrelated virtual-memory systems on separate nodes cannot provide.

Built-in virtual-memory support for data-parallel programming would allow the memory requirements of application programs to exceed the available memory size without increasing software development time or software complexity. Yet, programmers would not need specialized knowledge of I/O-optimal algorithms in order to avoid huge performance penalties.

This paper describes a linguistic step toward such a solution. Our approach is based on the data-parallel language C* [TMC93]. A preprocessor, ViC* (Virtual-memory C*), transforms a C* program with parallel variables so large that they must reside on disk into a C* program with all parallel variables fitting in memory and explicit I/O and library calls added. A ViC* source program does not declare individual variables as disk-resident, or out-of-core; instead, any C* shape may be declared to be `outofcore`, which means that all parallel variables of this shape are out-of-core. Linguistically, the only difference between a ViC* program and a "standard" C* program is the `outofcore` shape modifier. The explicit I/O calls added by ViC* are direct parallel disk reads and writes of sections of out-of-core parallel variables. The library calls added by ViC* are typically for operations requiring communication in out-of-core parallel variables, e.g., reductions, gets, and sends.

One principle of this project is to exploit existing languages and software as much as possible. Although it performs as much analysis as many compilers, ViC* is a preprocessor. A programmer using ViC* would first run the source program (say `prog.vic`), which contains `outofcore` shape declarations, through ViC*, which produces a standard C* program (say `prog.cs`). This C* program would then be run through the usual C* compiler and linked with special ViC* libraries (containing the out-of-core communication functions) using the compiler flag `-lvic` to produce an executable file.

Why choose C* as a base language? We want a data-parallel language that is used at several sites, for which there is existing code, and that is not High-Performance Fortran (HPF). We are interested in data-parallelism because it has proven to be a valuable parallel-programming paradigm and because recent I/O-optimal algorithms fit nicely into it. We want a language in use at several sites and for which there is existing code because we hope that scientific programmers and others will really use ViC*. And we wish to sidestep HPF because there are already two out-of-core HPF projects that we know of, at Syracuse University [BTC94, TBC94] and Rice University [KKP94], and because C* presents different implementation challenges from HPF. In particular, HPF uses arrays for parallelism and C* uses shapes, HPF has a much more restricted notion of context than C*, and HPF's style of data distribution differs from that of C*. Moreover, the ViC* run-time library may be substantially different from those in the out-of-core HPF projects. Despite recent setbacks at Thinking Machines Corporation, where C* was developed, we believe that C*—or at least the concepts behind it—will persist for some time to come.

The remainder of this paper is organized as follows. Section 2 presents a brief overview of the C* language and the ViC* extensions. The ViC* data structures and run-time interface reside in a header file, which Section 3 describes. Section 4 shows how ViC* modifies out-of-core shape and

data declarations. Sections 5 and 6 briefly discuss how ViC* transforms out-of-core `with` and `where` statements. Most of ViC*'s processing deals with expressions and optimizations, which Section 7 covers. Section 8 touches on issues relating to function calls with out-of-core parameters or return values. Section 9 offers some concluding remarks. Finally, Appendix A shows the C* code generated by a sample program.

2 Background concepts and overview of ViC*

This section introduces the parallel programming model and the language features of C* and ViC* that implement it. More information about the C* language appears in [TMC93].

C*, and hence ViC*, support *data-parallel programming*, in which a sequential program operates on parallel arrays of data, with each *virtual processor* operating on one parallel data element. The underlying computer multiplexes a set of physical processors among the virtual processors to support the parallel model. Scalar data remains global to all virtual processors.

Each parallel variable in C* has a *shape*, which describes its rectangular structure. A shape defines the logical configuration of parallel data in virtual processors. At any point in the program, a *current shape* is in force. Most parallel operations must operate on data of the current shape. A `with` statement selects the current shape, referred to by the reserved word `current`.

All C* operations are controlled by a *context*, which describes the active positions in parallel variables of the current shape. *Active positions* are those whose virtual processors initiate parallel operations. When a shape is made the current shape, all of its positions are active. The `where` statement narrows the context by selecting a subset of the active positions within the shape. An optional `else` clause of a `where` statement selects the complementary subset. The `everywhere` statement makes all positions active. Context has both static and dynamic qualities. It is static because completion of a `where` or `everywhere` statement restores the previous context. It is dynamic because it is implicitly passed to any called function.

Parallel *communication* transfers parallel data among the virtual processors. There are several forms of parallel communication. *Reductions* combine elements of a parallel variable into a scalar result. *Left indexing* a parallel variable stores or extracts a scalar in a single virtual processor. *Parallel left indexing* addresses data in a set of virtual processors. Virtual processors executing a *get operation* in an expression fetch data from other virtual processors. Virtual processors executing a *send operation* in an assignment transmit data to other virtual processors. Some readers may be more familiar with get as "gather" and with send as "scatter." The standard C* library includes specialized get and send operations for grid topologies as well as other forms of communication, notably scans.

C* is restricted to *in-core* data stored in the computer's main memory. ViC* processes large *out-of-core* parallel data sets stored on disk. ViC* extends the C* shape declaration statement by allowing the `outofcore` storage-class modifier on shape declarations. The `outofcore` modifier indicates that parallel variables of the shape in question normally reside outside the main memory. (Typically, `outofcore` variables will reside on a parallel disk system.) ViC* uses this information to access parallel variables of an `outofcore` shape. The keyword `outofcore` is a reserved word in our implementation. Currently, ViC* needs to know at compile time whether each shape (even those referenced by pointer) is `outofcore`. Eventually, we would like to be able to do without such a modifier and let ViC* determine which shapes should be out-of-core based on their size, the size of the main memory, and other related factors.

To translate out-of-core operations, ViC* decomposes a program into basic blocks. A *basic block* [ASU86] is a maximal sequence of statements with linear control flow. Types of statements

within a basic block are declarations, assignments, expressions, subroutine calls, as well as `with` and `where` statements. (Although `where` resembles a parallel `if`, it affects execution within virtual processors rather than control flow.) A directed acyclic graph (*dag*) represents data dependencies in a basic block. Basic blocks are further divided into simple blocks. A *simple block* is a subset of a basic block delimited by subroutine calls or communication within out-of-core shapes. Each simple block results in a *sectioning loop* in the C* program produced by ViC*. A simple block *strip-mines* [ZC90] out-of-core data by iterating over in-core *sections* or *strips* of the data.

To improve performance, ViC* uses *dataflow* analysis to determine which out-of-core data is required from each simple block in subsequent execution. Such data is *live*. A variable for which there are no further references is *dead*. Variables are always dead at the end of the block in which they are declared.

Sample program

We illustrate some of the above concepts with the ViC* program in Figure 1. The program declares an out-of-core shape `series`. Parallel variables of this shape are vectors of 2^{40} elements. The out-of-core parallel variable `normal` is a vector of 2^{40} floats. Within the function `example`, we declare out-of-core parallel variables `harmonic` and `k` in this shape. The variable `k` is initialized with the `pcoord` intrinsic function, which returns the index of each position along an axis of a shape. For $i = 0, 1, \ldots, \mathtt{N}-1$ the ith position of `k` (denoted `[`i`]k` in C*) has the value i. The `where` statement narrows the context to the positive positions. The first statement in the `where` body assigns elements of `harmonic` the reciprocals of `k`, so that `[`i`]harmonic` equals $1/i$, for $i = 1, 2, \ldots \mathtt{N}-1$. Narrowing the context to positive positions avoids a division by 0. A sum reduction, denoted by the `+=` operator, sums all elements of `harmonic` into `sum`. The series is normalized by the reciprocal of this sum into `normal`. Finally, a send operation shifts the elements of `normal`, sending each element to the previous position. Within the parallel left-index expression, the period denotes the index vector; hence this statement executes in parallel the assignments

```
[0]normal = [1]normal;
[1]normal = [2]normal;
[2]normal = [3]normal;
          ...
[N-2]normal = [N-1]normal;
```

3 Header File

The preprocessor includes a standard header file with every compilation. Figure 2 shows this file, which defines ViC* data structures and declares runtime support routines for memory-management and I/O. ViC* uses names beginning `ViC__` to avoid conflicts with the source program.

ViC* transforms out-of-core shape and data declarations into `ViC__shape` and `ViC__data` declarations. A `ViC__shape` structure describes an out-of-core shape. The `positions`, `rank`, and `dimensions` fields describe a parallel data configuration similar to an in-core shape, but without in-core size limitations. A shape also has a current context, represented here by a pointer to parallel boolean variable. A `ViC__data` structure describes a single out-of-core variable. It records the variable shape and identifies either a file with out-of-core data or a scalar initial value for a file yet to be created.

C* defines the current shape and context as global state, which is automatically passed to subroutines. ViC* implements these with global pointers to the current out-of-core shape and context. It transforms out-of-core `with` and `where` statements into operations on these pointers.

```
/* compute a truncated harmonic series and normalize it */

#define N   (1L<<40)                        /* N == 2**40 */

outofcore shape [N]series;                  /* series of terms */
float:series    normal;                     /* normalized terms */

void example()
{
  float:series    harmonic;                 /* truncated harmonic series */
  float           sum;                      /* series summation */
  int:series      k;

  with (series) {
    k = pcoord(0);                          /* index array */
    where (k > 0) {
      harmonic = 1.0 / k;                   /* 1/k (0 < k < N) */
      sum = += harmonic;                    /* sum terms */
      normal = harmonic * (1.0 / sum);      /* normalized terms */
      [.-1]normal = normal;                 /* shift series */
    }
  }
}
```

Figure 1: A sample ViC* program.

Other ViC* runtime support subroutines (`ViC__Open` through `ViC__Copy` in Figure 2) allocate in-core strips of out-of-core data and transfer these strips to and from data files. The routines determine the in-core shape at runtime to make the best use of available memory.

ViC* converts out-of-core operations into loops that iterate over in-core sections from disk files. Several macros (`ViC__DCL` through `ViC__CLOSE` in Figure 2) simplify declarations and file access. These macros use the standard preprocessor token-concatenation operator `##` to reference auxiliary variables. Variables of the form `ViC__`*name*`_file` and `ViC__`*name*`_strip` reference open file descriptors and in-core data. ViC* also introduces temporary variables beginning `ViC__`*number* to hold stacked context information and intermediate results.

Other library functions implement out-of-core parallel communication, with versions for a number of special cases [Cor92].

4 Processing of declarations

This section describes how ViC* processes out-of-core shape and data declarations. ViC* identifies out-of-core shape declarations by the `outofcore` modifier and replaces them with `ViC__shape` declarations. It scans variable declarations for these shapes to identify out-of-core data and replaces them with `ViC__data` declarations. ViC* detaches any out-of-core initializations for separate processing as assignments, and it copies other declarations unchanged to the output. Figure 3 shows several typical declarations, with the corresponding ViC*-generated output.

The out-of-core shape `series` has `N` positions in a 1-dimensional array. Its initial context is the parallel boolean variable `ViC__everywhere`. The declarations of out-of-core variables `a`, `aa`, and `ap` are changed into declarations using `ViC__data` structures with shape `series`. The array `aa` is initialized with an array of `ViC__data` structures. These out-of-core variables have no initial value.

The data for an out-of-core variable is kept in a file identified by the `ViC__data` structure's

```
#define ViC__MAX_DIM  8

/* ViC outofcore descriptors */
struct ViC__shape {                         /* out of core shape */
  struct ViC__data *context;                /* pointer to the saved context */
  size_t      positions;                    /* the number of shape positions */
  int         rank;                         /* shape rank */
  size_t      dimensions[ViC__MAX_DIM];     /* number of elements in each dimension */
};
typedef struct ViC__shape ViC__shape;

struct ViC__data {                          /* outofcore dataset */
  ViC__shape  *datashape;                   /* pointer to the variable shape */
  long        fileid;                       /* data file ID */
  void        *initial;                     /* or static initial value */
};
typedef struct ViC__data ViC__data;

/* ViC global "registers" passed implictly to subroutines */
ViC__shape  *ViC__current;                  /* current shape */
ViC__data   *ViC__context;                  /* current context */

/* simple block processing */
shape       *ViC__Stripshape(size_t sizes);  /* determine strip size & create shape */
void        *ViC__Freeshape(shape *s);       /* release shape for strip */
void:void   *ViC__Alloc(size_t size);        /* allocate in-core strip */
void        ViC__Free(void);                 /* free in-core strips */

/* out-of-core I/O */
typedef struct ViC__file ViC__file;         /* data file descriptor */

ViC__file   *ViC__Open(ViC__data *data);                  /* open dataset */
void        ViC__Close(ViC__file *file, ViC__data *data);  /* close dataset */
void        ViC__Readstrip(void:void *data, size_t size, ViC__file *file);
void        ViC__Writestrip(void:void *data, size_t size, ViC__file *file);
void        ViC__Remove(ViC__data *data);                 /* delete file out of scope */
void        ViC__Copy(ViC__data *dst, ViC__data *src);    /* copy variables */

/* constants */
extern ViC__data    ViC__everywhere;        /* everywhere context */

/* macros to declare/read/write strip data */
#define ViC__DCL(T, V, D)                                                   \
    T:current *ViC__##V##_strip = ViC__Alloc(boolsizeof(ViC__##V##_strip));  \
    ViC__file *ViC__##V##_file = ViC__Open(D)
#define ViC__LOOP                                   \
    long ViC__position;                             \
    for (ViC__position = 0;                         \
         ViC__position < ViC__current->positions;  \
         ViC__position += positionsof(*ViC__stripshape))
#define ViC__READ(V)    \
    ViC__Readstrip(ViC__##V##_strip, boolsizeof(ViC__##V##_strip), ViC__##V##_file)
#define ViC__WRITE(V)   \
    ViC__Writestrip(ViC__##V##_strip, boolsizeof(ViC__##V##_strip), ViC__##V##_file)
#define ViC__CLOSE(V, D)      ViC__Close(ViC__##V##_file, D)
```

Figure 2: The ViC.h file.

```
outofcore shape [N]series, *sp;  ===>   ViC__shape      series  = { &ViC__everywhere, N, 1, {N} },
                                                       *sp;

float:series    a, aa[5], *ap;   ===>   ViC__data       a       = { &series, 0, NULL },
                                                       aa[5]   = { { &series, 0, NULL },
                                                                   { &series, 0, NULL },
                                                                   { &series, 0, NULL },
                                                                   { &series, 0, NULL },
                                                                   { &series, 0, NULL } },
                                                       *ap;

float:*sp       x;               ===>   ViC__data           x = { sp, 0, NULL };

static float:series y = 1.0;     ===>   static float        ViC__10 = 1.0;
                                        static ViC__data    y = { &series, 0, &ViC__10 };

struct s {                       ===>   struct s {
  int f1;                                 int f1;
  float f2;                               float f2;
  char f3;                                char f3;
};                                      };
struct s:series z;                      struct ViC__s {
                                          ViC__data f1 = { &series, 0, NULL };
                                          ViC__data f2 = { &series, 0, NULL };
                                          ViC__data f3 = { &series, 0, NULL };
                                        };
                                        struct ViC__s z;
```

Figure 3: Some sample ViC* declarations and the corresponding output.

`fileid` field. Initially a variable has no value and a zero `fileid`. Upon the first assignment to a variable, the run-time function `ViC__Writestrip` assigns a unique nonzero integer to `fileid` and creates a file with that name. This file holds the variable's data for the lifetime of the variable.

Global and static variables may be assigned an initial value before the program runs via the `ViC__data` structure's `initial` pointer, which points to a scalar initial value. The static out-of-core variable `y` has an initial value `1.0` stored in a preprocessor-generated scalar `ViC__10`. Global out-of-core data files are deleted when the program exits.

There are two obvious ways to store parallel structures. One way, used by ViC*, places each field in a separate parallel variable. An element of a parallel structure, therefore, does not exist as a contiguous entity. The other way keeps the fields of each element together, storing them in a single parallel variable. Such an implementation generally requires maintaining a stride value and field offsets in order to access individual fields. ViC* places each field in a separate parallel variable for two reasons. First, addresses of fields of parallel structs are expressible values in C*, but addresses of individual elements of parallel variables of any type are not. Using the declarations in Figure 3, for example, `&z.f2` is a legal C* expression but `&[5]z` is not. ViC*'s strategy is oriented toward the former; the expression `&z.f2` in a ViC* source program would remain unchanged in the output. Second, individual fields within structures are expressible values in C*, and so are entire structures. Thus, the C* statements `a = z.f2` and `struct s zz = [5]z` are both legal. ViC* can implement the first statement easily, and it can implement the second statement by reading a single element from each of three parallel variables. The opposing strategy of keeping fields together allows the second statement to be easily implemented. To implement the first statement, however, it must read through more data, most of which is unneeded, and hence it entails many more disk reads

```
{
  ViC__shape *ViC__prev = ViC__current;   /* save previous shape in temp */
  ViC__current->context = ViC__context;   /* save context */
  ViC__current = &SHAPE;                  /* point to current ViC__shape */
  ViC__context = ViC__current->context;   /* take its context */

  WITHBODY;                               /* body of with */

  ViC__current = ViC__prev;               /* restore previous shape */
  ViC__context = ViC__current->context;   /* and context */
}
```

Figure 4: Code generated for the `with (SHAPE) WITHBODY` statement.

than the ViC* strategy.

5 Processing of `with` statements

This section outlines how ViC* processes `with` statements. A `with` statement establishes the current shape and sets the context to the current context for that shape. C* allows `with` statements to be nested, with the shape and context restored to their previous values after the end of the statement.

ViC* transforms a `with` statement that establishes an out-of-core shape into assignments to `ViC__current` and `ViC__context`. A temporary variable saves the previous shape and restores it after the `with` statement. ViC* defers output of a `with` statement until it encounters an out-of-core operation in the body of the `with`. The function `ViC__Stripshape` computes an in-core shape used to hold in-core strips of the data.

Figure 4 shows the structure of the code generated for an out-of-core `with` statement of the form

```
with (SHAPE) WITHBODY;
```

Here a local temporary variable, `ViC__prev`, stacks the previous out-of-core shape. The statement body, `WITHBODY`, undergoes further processing.

6 Processing of `where` statements

This section describes how ViC* processes `where` statements. The `where` statement restricts the execution context for a statement. The `everywhere` variant of this statement widens the context to include all positions. This execution context determines which positions initiate parallel operations. Context is implicit in all parallel operations, including those in function calls. This use of a current context is one of the distinguishing features of C*, in contrast to FORTRAN 90 [ANS92, MR90], where context has static scope and may be applied only to parallel operations.

Like the `with` statement, a `where` statement saves state and restores it. ViC* replaces out-of-core `where` statements with operations on out-of-core boolean variables and the current context pointer, `ViC__context`. It copies the current context, evaluates the `where` expression in the surrounding context, then computes the restricted context. For the `else` clause, the value of the `where` expression is complemented to narrow the context.

Figures 5 and 6 show the structure of the code generated for out-of-core `where` statements

```
  {
@   bool:outofcore current  *ViC__prev,     /* saved context pointer */
                            ViC__new,       /* new context data */
                            ViC__where;     /* where expression */
    ViC__prev = ViC__context;               /* save pointer to old */

@   ViC__where = CONTEXT;                   /* evaluate where expression */
@   everywhere
@     ViC__new = *ViC__prev & ViC__where;   /* copy old context data */
    ViC__context = &ViC__new;               /* make current */

@   WHEREBODY;                              /* execute body of where */

@   everywhere
@     ViC__new = *ViC__prev & !ViC__where;  /* recopy old context */

@   ELSEBODY;                               /* execute body of where else */

    ViC__context = ViC__prev;               /* restore previous context */
  }
```

Figure 5: The ViC* code for the statement `where (CONTEXT) WHEREBODY; else ELSEBODY`. Lines beginning with `@` undergo further processing by ViC*.

```
  {
@   bool:outofcore current  *ViC__prev;     /* saved context pointer */
    ViC__prev = ViC__context;               /* save pointer to old */

    ViC__context = &ViC__everywhere;        /* expand context */

@   EVERYBODY;                              /* body of everywhere */

    ViC__context = ViC__prev;               /* restore previous context */
  }
```

Figure 6: The ViC* code for the statement `everywhere EVERYBODY`. Lines beginning with `@` undergo further processing by ViC*.

```
where (CONTEXT)
    WHEREBODY;
else
    ELSEBODY;
```

and

```
everywhere EVERYBODY;
```

in terms of ViC* rather than C* code. ViC* does further processing on the context assignments and the statements in `WHEREBODY`, `ELSEBODY`, and `EVERYBODY`. The assignments to context variables are structured so that they may be combined with other out-of-core operations in the statement body. The scope of these variables is limited to the `where` statement, so that they need not be written to disk if the statement fits in a simple block. Some of the variables can be eliminated if there is no `else` part or for `everywhere` statements. Section 7 describes these and other optimizations.

7 Processing of expressions

This section describes how ViC* processes expressions, including C* assignments. Because ViC* is a front end to a C* compiler, it is only concerned with translating the `outofcore` extensions to C*. Out-of-core operations consist of parallel operations, reductions, and left indexing of `outofcore` variables. The `outofcore` syntax extensions ensure that all such operations can be identified in the source. Scalar C expressions and expressions involving in-core parallel variables are compiled by the C* compiler, and so they need no additional processing.

ViC* uses the current out-of-core shape and context to generate C* code that evaluates out-of-core expressions by making passes in sectioning loops. Each sectioning loop evaluates a series of in-core sections, or strips. The loop prologue evaluates any scalar expressions used in the body of the loop.

The loop body transfers data between out-of-core data files and the in-core strips and evaluates out-of-core expressions in the appropriate context. A loop may cover several contexts, that is, several `where` or `everywhere` statements.

Elementwise expressions

Let us first examine a simple elementwise expression:

```
harmonic = 1.0 / k;
```

becomes

```
{
  shape *ViC__stripshape = ViC__Stripshape(              /* determine strip shape */
                             boolsizeof(bool:current) +
                             boolsizeof(float:current) +
                             boolsizeof(int:current));

  with (*ViC__stripshape) {                              /* in-core shape */
    ViC__DCL(bool, 30, ViC__context);                    /* in-core context */
    ViC__DCL(float, harmonic, &harmonic);                /* in-core data */
    ViC__DCL(int, k, &k);

    ViC__LOOP {
      ViC__READ(30);                                     /* in-core context */
      ViC__READ(k);                                      /* in-core data */
      where (*ViC__30_strip) {                           /* establish context */
        *ViC__harmonic_strip = 1.0 / *ViC__k_strip;      /* evaluate expression */
      }
      ViC__WRITE(harmonic);                              /* write output data */
    }

    ViC__Free();                                         /* free in-core strips */
    ViC__CLOSE(k, &k);                                   /* close data files */
    ViC__CLOSE(harmonic, &harmonic);
    ViC__CLOSE(30, ViC__context);
  }

  ViC__Freeshape(ViC__stripshape);                       /* delete shape */
}
```

The function `ViC__Stripshape` computes and allocates the in-core shapes, given the sizes of the strip data and the context. The `ViC__DCL` macro declares and allocates the in-core data and then opens the data files. The `ViC__READ` macro reads data and context for the out-of-core expression,

with results written by `ViC__WRITE`. Once the loop completes, the files are closed and the shape and data are deallocated.

Reductions

A reduction operator, such as `+=`, accumulates its result in a scalar variable. This reduction value is available at the end of the loop. The loop for

```
sum = += harmonic;
```

with float scalar `sum` becomes

```
{
  shape *ViC__stripshape = ViC__Stripshape(   /* determine strip shape */
                           boolsizeof(bool:current) +
                           boolsizeof(float:current));

  with (*ViC__stripshape) {                   /* in-core shape */
    float ViC__40 = 0.0;                      /* scalar temporary */
    ViC__DCL(bool, 30, ViC__context);         /* in-core context */
    ViC__DCL(float, harmonic, &harmonic);     /* in-core data */

    ViC__LOOP {
      ViC__READ(30);                          /* in-core context */
      ViC__READ(harmonic);                    /* in-core data */
      where (*ViC__30_strip) {                /* establish context */
        ViC__40 += *ViC__harmonic_strip;      /* evaluate reduction */
      }
    }

    ViC__Free();                              /* free in-core strips */
    ViC__CLOSE(harmonic, &harmonic);          /* close data files */
    ViC__CLOSE(30, ViC__context);
  }

  ViC__Freeshape(ViC__stripshape);            /* delete shape */
}

sum = ViC__40;                                /* result of reduction */
```

Optimization by loop fusion

ViC* attempts to reduce the amount of I/O by combining sectioning loops that share inputs and outputs. If the two expressions above are combined, their sectioning loops are *fused* into a single loop. The body of the fused sectioning loop for the code fragment

```
k = pcoord(0);
where (k > 0) {
  harmonic = 1.0 / k;
  sum = += harmonic;
}
```

is transformed into the following, which computes a float scalar `ViC__40`:

```
*ViC__k_strip = ViC__position + pcoord(0);       /* evaluate expression */
*ViC__30_strip = (*ViC__k_strip > 0)
where (*ViC__30_strip) {
  *ViC__harmonic_strip = 1.0 / *ViC__k_strip;    /* evaluate expression */
  ViC__40 += *ViC__harmonic_strip;              /* evaluate reduction */
}
```

Note that the body of the loop computes the temporary variable used to hold context information, eliminating the need to read it from disk.

The optimizer operates on basic blocks containing out-of-core operations. A basic block consists of expressions and assignments with linear control flow. Data dependencies within a basic block form a dag. Vertices identify the results of the operations in the block, and edges connect operands to an operation. Any in-order traversal of the dag correctly evaluates the operations.

The dag is used to partition the basic block into subgraphs, each of which is a simple block, implemented with a sectioning loop. Within a subgraph, initial vertices are those which have no input edges, such as variable values. Final vertices are those which have no output edges, such as assignments to variables whose value is not used in the subgraph. Operations that potentially involve communication, such as reductions, left indexing, and subroutine calls, delimit simple blocks and must be either initial or final vertices. They are final vertices of the simple block that computes them, and they are initial vertices of the simple block that uses them. Such values are implemented with temporary variables that transmit their value from one block to another.

In the sample program above, we combine the parallel division and summation into one simple block, but the result of the summation is not available until the next simple block. When we include the dependency on `sum` in the program

```
harmonic = 1.0 / k;
sum = += harmonic;
normal = harmonic * (1.0 / sum);
```

the assignment to `normal` requires a second sectioning loop.

Likewise, the vertex for a `volatile` operand or one that is aliased must be initial or final. In addition, for debugging purposes, a compiler flag forces every out-of-core operation into a separate simple block. In this case, no loop fusion occurs and debugging can proceed statement by statement.

Observe that context, which we can view as an implicit operand of every expression (except in the scope of an explicit `everywhere`), motivates loop fusion even when it would otherwise be unprofitable. For example, in the absence of context, if two consecutive statements had no variables in common, we would avoid fusing their sectioning loops in order to maximize the in-core strip size. Larger strip sizes lead to faster I/O. If the statements share a context, however, fusing the loops obviates the need to read the context for the second statement and possibly even the need to write it to disk.

The optimizer also expands certain out-of-core constant expressions, such as the value `pcoord(0)` assigned to `k` above. Expressions expanded by the optimizer include simple arithmetic operations involving scalar constants and the intrinsic function `pcoord`.

An additional optimization avoids writing out-of-core variables that are no longer live, i.e., those that will not be read back. This optimization requires ViC* to perform a data flow analysis beyond the scope of a basic block.

```
outofcore shape [N]series;              ===>  ViC__shape series = { &ViC__everywhere, N,1,{N} };

float mean(float:series a);             ===>  float mean(ViC__data a);

float:outofcore current normalize(float:outofcore current a);
                                        ===>  void normalize(ViC__data *ViC__result, ViC__data a);

int:series calc(outofcore shape s);     ===>  void calc(ViC__data *ViC__result, ViC__shape s);
```

Figure 7: Example function headers and the result of their processing.

8 Processing of functions and communication

This section describes how ViC* processes function declarations, parameters, and results. ViC* processes function definitions and function calls with out-of-core formal parameters and results; in-core parameters and results remain unchanged. Parameter and result declarations are transformed as described in Section 4. Statements in the function body are rewritten to operate on out-of-core data as described in Section 7. Function calls are processed to pass arguments and return results by value, in conformance to the C and C* convention.

Out-of-core shape and data parameters are replaced with the corresponding `ViC__shape` and `ViC__data` declarations, as shown in Figure 7. Out-of-core shape parameters and parameters using the shape `current` must be declared with the `outofcore` keyword. ViC* ensures that out-of-core data that is passed by value is copied by the caller. This copying may be combined in a simple block with other operations in the caller, or it can be achieved by calling a library subroutine, `ViC__Copy`, to copy the data.

Instead of returning an out-of-core result by value, ViC* passes a pointer to an out-of-core result variable as an extra parameter. Within the function body, all `return` statements copy the return value to the result parameter. This assignment may be combined in a simple block with other computations. By using a result parameter, ViC* avoids an additional data copy by the caller.

Local variables and temporaries must be deleted when the function exits or returns. In-core data is allocated on the stack and is automatically reclaimed. Out-of-core data is persistent, and must be explicitly deleted by calling `ViC__Remove` for each local variable at every exit point.

It is generally more efficient to pass out-of-core parallel data by reference, using pointers, than by value, which would entail copying the data. Admittedly, this style may increase the programming effort required to use out-of-core data. In some cases, however, different algorithms are more efficient for out-of-core data, making a distinct out-of-core version desirable. Function overloading in C* allows both versions to coexist.

ViC* processes all communication that involves out-of-core variables (accomplished by parallel left indexing) by generating calls to run-time library functions. This run-time library is an integral part of the virtual-memory package and implements efficient algorithms for special cases of communication [Cor92]. The ViC* library implements left indexing of out-of-core variables where the index is scalar, in-core parallel, or out-of-core. It also implements left indexing of in-core parallel variables when the index is out-of-core. The C* communication routines `to_grid`, `to_torus`, `from_grid`, and `from_torus` are also overloaded to support out-of-core data, such as the send operation `[.-1]normal = normal` in the sample program.

9 Conclusion

ViC* supports parallel programming on large data sets. It takes advantage of the distributed memory model of C* to provide the convenience of in-core programming for data too large to fit in main memory. With a small extension to the C* language, this approach implements virtual memory for parallel data. ViC* gives C* programmers a migration path for applications that exhaust physical memory resources.

ViC* goes beyond a simple library interface. For large datasets, we expect the time spent performing I/O to exceed the time spent computing. ViC* analyzes program data flow and performs program transformations to reduce I/O demands. It calculates the amount of in-core data to make full use of available memory. ViC* fuses in-core sectioning loops to avoid repeated transfers of the same data. These loops can cross context boundaries. In some cases, ViC* recomputes expressions to avoid saving and recalling their values.

In addition, we expect ViC* to serve as a testbed for parallel I/O research. Because I/O is under the ViC*'s control, we can experiment with parallel I/O algorithms and implementations. Areas of investigation include tradeoffs between synchronous and asynchronous I/O [KKP94], out-of-core algorithms [Cor92], and parallel file systems [CK93].

Other extensions to ViC* may include optimizations of *elemental* functions [KF93]—those which require no parallel communication. We can evaluate an elemental function on an in-core strip, calling it within the body of a sectioning loop. For arbitrary communications using left indexing, we expect to categorize index values where possible by communications topology, such as grid, torus, transpose, and general permutation. There are special algorithms [Cor92] for these communication patterns.

File systems normally support persistent data. Program data, such as ViC* out-of-core data, is transient and disappears on program exit. ViC* ensures that such data is properly deleted on scope exit. Global out-of-core data must be deleted specially. Alternatively, ViC* out-of-core data might be used to support persistent variables or program checkpointing.

Acknowledgments

Georgi Vassilev participated in the early stages of ViC* design. We thank Phil Hatcher for access to the source code of his C* compiler and for his comments. Thanks also to Jamie Frankel, David Gingold, and Dave Loshin for helpful discussions about C*.

References

[ANS92] American National Standards Institute, Inc. *X3.198-1992 Fortran—Extended*, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BTC94] Rajesh Bordawekar, Rajeev Thakur, and Alok Choudhary. Efficient compilation of out-of-core data parallel programs. Technical report, NPAC, April 1994.

[CGG+94] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms (extended abstract). Submitted to SODA '95, July 1994.

[CK93] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the DAGS '93 Symposium*, pages 64–74, June 1993. Also available as Dartmouth College Computer Science Technical Report PCS-TR93-188.

[Cor92] Thomas H. Cormen. *Virtual Memory for Data-Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992. Available as Technical Report MIT/LCS/TR-559.

[Cor93] Thomas H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, January and February 1993.

[CSW94] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Department of Computer Science, July 1994. Submitted to *IEEE Transactions on Parallel and Distributed Systems*. Preliminary version appeared in Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures.

[Den70] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[GTVV93] Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, November 1993.

[KF93] Alexander C. Klaiber and James L. Frankel. Comparing data-parallel and message-passing paradigms. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II:11–II:20, August 1993.

[KKP94] Ken Kennedy, Charles Koelbel, and Mike Paleczny. Scalable I/O for out-of-core structures. Technical Report CRPC-TR93357-S, Center for Research on Parallel Computation, Rice University, August 1994.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposiumn on Operating Systems Design and Implementation*, November 1994. Available as Dartmouth College Department of Computer Science Technical Report PCS-TR94-226.

[MR90] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford University Press, 1990.

[NV91] Mark H. Nodine and Jeffrey Scott Vitter. Large-scale sorting in parallel memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 29–39, July 1991.

[NV92] Mark H. Nodine and Jeffrey Scott Vitter. Optimal deterministic sorting on parallel disks. Technical Report CS-92-08, Department of Computer Science, Brown University, 1992.

[TBC94] Rajeev Thakur, Rajesh Bordawekar, and Alok Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. In *IPPS '94 Workshop on Parallel I/O*, April 1994.

[TMC93] Thinking Machines Corporation. *C* Programming Guide*, May 1993.

[VS94] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.

[WGWR93] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *DAGS '93*, June 1993.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press and Addison-Wesley, 1990.

A ViC* Output

This appendix shows the code ViC* generates for the sample program of Figure 1. Comments are added for clarity; ViC* normally only copies comments unchanged.

```
#define     N   (1L<<40)                                  /* N == 2**40 */

ViC__shape  series  = { &ViC__everywhere, N, 1, {N} };
ViC__data   normal = { &series, 0, NULL};

void example()
{
  ViC__data   harmonic = { &series, 0, NULL };
  float       sum;
  ViC__data   k = { &series, 0, NULL };

  /* with (series) */
  {
    ViC__shape  *ViC__prev = ViC__current;              /* save previous shape */
    ViC__current->context = ViC__context;               /* save context */
    ViC__current = &series;                             /* point to current ViC__shape */
    ViC__context = ViC__current->context;               /* take its context */

    /* generate harmonic series and compute sum
       harmonic = 1.0/k;
       sum = += harmonic;
       */
    {
      shape *ViC__stripshape = ViC__Stripshape(         /* determine strip shape */
                               boolsizeof(bool:current) +
                               boolsizeof(float:current) +
                               boolsizeof(int:current));

      with (*ViC__stripshape) {                         /* in-core shape */
        ViC__DCL(bool, 10, ViC__context);               /* in-core context */
        ViC__DCL(float, harmonic, &harmonic);           /* in-core data */
        ViC__DCL(int, k, &k);
        float ViC__40 = 0.0;                            /* reduction accumulator */

        ViC__LOOP {
          ViC__READ(10);

          where (*ViC__10_strip) {
            *ViC__k_strip = ViC__position + pcoord(0);
          }

          where (*ViC__10_strip & ViC__k_strip > 0) {   /* establish context */
            *ViC__harmonic_strip = 1.0 / *ViC__k_strip;
            ViC__40 += *ViC__harmonic_strip;
          }
          ViC__WRITE(harmonic);                         /* write output data */
        }

        ViC__Free();                                    /* free in-core strips */
        ViC__CLOSE(k, &k);                              /* close data files */
        ViC__CLOSE(harmonic, &harmonic);
        ViC__CLOSE(10, ViC__context);                   /* close context */
        sum = ViC__40;                                  /* reduction operation */
      }
```

```
      ViC__Freeshape(ViC__stripshape);                  /* delete shape */
    }

    /* use sum to generate normalized series
       normal = harmonic*(1.0/sum);
       */
    {
      shape *ViC__stripshape = ViC__Stripshape(         /* determine strip shape */
                               boolsizeof(bool:current) +
                               boolsizeof(int:current) +
                               boolsizeof(float:current) +
                               boolsizeof(float:current));

      with (*ViC__stripshape) {                         /* in-core shape */
        ViC__DCL(bool, 12, ViC__context);
        ViC__DCL(int, k, &k);
        ViC__DCL(float, harmonic, &harmonic);           /* in-core data */
        ViC__DCL(float, normal, &normal);

        ViC__LOOP {
          ViC__READ(12);                                /* in-core context */
          ViC__READ(harmonic);                          /* in-core data */

          where (*ViC__12_strip) {
            *ViC__k_strip = ViC__position + pcoord(0);
          }

          where (*ViC__12_strip & *ViC__k_strip > 0) {  /* establish context */
            *ViC__normal_strip = *ViC__harmonic_strip * (1.0 / sum);
          }
          ViC__WRITE(normal);                           /* write output data */
        }

        ViC__Free();                                    /* free in-core strips */
        ViC__CLOSE(normal, &normal);                    /* close data files */
        ViC__CLOSE(harmonic, &harmonic);
        ViC__CLOSE(k, &k);
        ViC__CLOSE(12, ViC__context);                   /* close context */
      }

      ViC__Freeshape(ViC__stripshape);                  /* delete shape */
    }

    to_grid(&normal, &normal, &normal, -1);             /* grid communication */

    ViC__current = ViC__prev;                           /* restore previous shape */
    ViC__context = ViC__current->context;               /* and context */
  }
}
```