Dartmouth College Computer Science Technical Report
PCS-TR97-323
November 1997

ViC*: A Compiler for Virtual-Memory C*

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

Abstract

This paper describes the functionality of ViC*, a compiler for a variant of the data-parallel language C* with support for out-of-core data. The compiler translates C* programs with shapes declared `outofcore`, which describe parallel data stored on disk. The compiler output is a SPMD-style program in standard C with I/O and library calls added to efficiently access out-of-core parallel data. The ViC* compiler also applies several program transformations to improve out-of-core data layout and access.

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.

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

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

†Supported by the National Science Foundation under Grant CCR-9625894. Author's email address: `thc@cs.dartmouth.edu`.

This paper is a major update of [CC94].

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]. Today's parallel machines typically run traditional sequential virtual memory on the individual nodes. 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 I/O performance in *out-of-core* problems, i.e., those whose data requirements exceed the size of main memory.

There are multiple paths to reducing I/O times in out-of-core computations. One way is to make each disk access faster; this approach is beyond the scope of the this paper and the ViC* project. Another way is to reduce the number of disk accesses. ViC* is based on this approach.

We know of two ways to reduce the number of disk accesses in an out-of-core data-parallel computation, and the ViC* project uses them both. One is to have the compiler transform the program into one that eliminates many of the disk accesses. The resulting program is essentially the same as the original, but improved. This approach is the focus of this paper.

The other way to reduce the number of disk accesses is to design algorithms that explicitly work with out-of-core data on parallel disks. Since the introduction of the Parallel Disk Model (PDM) by Vitter and Shriver in 1990 [VS94], there have been significant technical advances on how to carefully plan parallel disk accesses for common data-parallel operations and algorithms [AP94, Arg95, AVV95, BGV97, CGG+95, Cor93, Cor97, CN96, CWN97, CSW94, GTVV93, NV93, NV95, Wis96, WGWR93, VS94]. 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.

The ViC* approach of built-in virtual-memory support for data-parallel programming allows the memory requirements of application programs to exceed the available memory size without increasing software development time or software complexity. The ViC* compiler transforms the source program to remove many of the parallel disk accesses, and the ViC* runtime system invokes efficient PDM algorithms to perform specific tasks. Programmers do not need specialized knowledge of PDM algorithms in order to avoid huge performance penalties.

To be more specific, the ViC* system is based on using a data-parallel language, in particular C* [TMC93]. The ViC* (Virtual-memory C*) compiler transforms a C* program with parallel variables so large that they must reside on disk into a C program with I/O and library calls to access out-of-core data on a parallel disk system. 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. The I/O calls added by ViC* read and write sections of out-of-core parallel variables. Before emitting the C code, however, the compiler transforms the program to eliminate many of the I/O calls. The library calls added by ViC* are typically for operations requiring communication in out-of-core parallel variables, e.g., reductions, gets, and sends.

This paper focuses on the ViC* compiler, rather than on the library calls. The library calls are where the efficient PDM algorithms are invoked, and so the library is an important part of the full ViC* system. The compiler, therefore, yields two benefits. First, the transformations it applies directly reduce the number of I/O calls. Second, it makes calls, and enables the programmer to make calls, to the ViC* library, which further reduces the I/O costs.

One principle of this project is to exploit existing languages and software as much as possible. Rather than design a new language, ViC* implements an existing language, C*, with minor extensions. It produces C code, which is processed by host machine compilers.

Why choose C* as a base language? We want an established data-parallel language 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. C* presents different implementation challenges from HPF. In particular, HPF uses arrays, an existing language feature, for parallelism. On the other hand, C* uses shapes, a separate feature not found in sequential C. HPF specifies data distribution at compile time, whereas C* (and ViC*) evaluate shapes at runtime. C* also faces issues of pointer aliasing not permitted in HPF. Many of the issues raised in ViC* implementation are not particular to C*, but are common to compiled data-parallel languages. For example, the language F^{--} [dDEF+97] bears many similarities to C*. Finally, although many people think of C* solely as a bygone product of Thinking Machines Corporation, there is an active project under the direction of Phil Hatcher at the University of New Hampshire that has produced a C* compiler and runtime system for a distributed-memory model (see [LH92] and http://www.cs.unh.edu/pjh/cstar/cstar.html).

The remainder of this paper is organized as follows. Section 2 describes virtual memory and its implementation on parallel disk systems. Section 3 presents a brief overview of the C* language and the ViC* extensions. Section 4 discusses program transformations to improve access to out-of-core data, and Section 5 describes parallel data layout. Section 6 describes the runtime interface used to access out-of-core data. Section 7 presents performance measurements. We conclude in Section 8.

2 Virtual Memory

As described by Denning [Den70] in 1970, *virtual memory* presents the programmer "the *illusion* that he has a very large main memory at his disposal, even though the computer actually has a relatively small main memory." Demand paging is a common implementation of virtual memory, but, as we are about to see, for large data sets there are more efficient alternatives.

Demand paging

Demand paging is a runtime-only mechanism, implemented in the operating system with architectural support and requiring no language or compiler support. Pages are loaded into main memory on demand, i.e., when they are accessed. The program cannot proceed until

the data become available. Demand paging services page faults one at a time, based on accesses in a sequential program. It is well suited to a multiprogramming environment that emphasizes throughput rather than latency, since when one process blocks while waiting for page-fault service, another process can run.

Traditional demand paging has relatively poor performance when several passes are made over the same out-of-core data. A typical demand pager replaces the least recently used page (or at least a not very recently used page) with new pages. By the time the last page of out-of-core data is loaded, the first page has been replaced. The result is that each page must be reloaded for each pass through the data.

Just as some optimizing compilers do for in-core data that does not fit in cache, an out-of-core computation can be restructured to combine multiple passes. Even with restructuring to combine multiple passes, with demand paging the speed of the remaining passes is limited by access time to the swap area.

The swap area is typically a partition of a single disk. The access time can be reduced by using a parallel disk system to increase the data transfer rate. Even with the highly unusual configuration of a parallel disk system for swap space, only the transfer rate improves. I/O latency for demand paging does not improve.

I/O latency can be hidden in many out-of-core computations by prefetching and post-writing (see [CH97] for an example). How good would a demand paging system with a restructuring compiler, parallel disk system, and prefetching/post-writing be? If all computations made sequential passes over the data, it would be quite good.

However, some asymptotically optimal out-of-core algorithms for the Parallel Disk Model (e.g., those for sorting [BGV97, NV93, NV95, VS94], structured permutations [CSW94, Wis96], and FFTs [Cor97, CN96, CWN97]) do not access out-of-core data in a simple, sequential fashion. They read and write whole disk blocks, but the blocks may be scattered throughout the parallel disk system. These algorithms require the ability to independently access individual disks. Without an explicit I/O interface, they cannot take full advantage of a parallel disk system. To our knowledge, no demand paging system provides such control.

The principal advantage of demand paging is transparency, not performance. The program takes no part in the managing the virtual memory; indeed it is typically unable to determine that there is virtual memory. For small working sets, demand paging delivers adequate performance.

Virtual memory with ViC*

To support large, out-of-core working sets, ViC* sacrifices transparency for performance, with the program explicitly managing its own virtual memory. ViC* maintains near-transparency in the program source, however, requiring only the addition of `outofcore` specifications.

ViC* implements virtual memory for parallel data only; traditional virtual memory mechanisms are adequate to handle instructions and scalar data. Our approach is based on a combination of language features, compiler, and runtime support. We do not require, but

can take advantage of, operating system and architectural support for parallel disk systems. Our assumption is that parallel data sets are large enough to warrant special treatment in software. Each access deals with a large amount of data. In fact, the transfer size of a ViC* access is typically much larger than in a traditional demand paging system. Each ViC* access gets at least as much data per disk as in demand paging (if not more), and each ViC* access is to multiple disks. Consequently, the access cost in ViC* is spread over many more elements than in traditional demand paging.

In this paper we concentrate on data-parallel operations, where the access pattern is the same for all elements. For such code the compiler is able to exploit its knowledge of program structure to reorder accesses, reducing the number of page transfers to and from main memory.

As mentioned previously, ViC* also includes a library of optimal out-of-core algorithms for permutations and other data movement. These algorithms take advantage of an independent I/O interface to a parallel disk system. Such algorithms offer large speedups over conventional in-core algorithms under demand paging. In one case, an explicit out-of-core FFT algorithm was over 144 times faster than a demand-paged version of the traditional in-core Cooley-Tukey method [CN96].

3 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*, supports *data-parallel programming*, in which a sequential program operates on parallel data distributed among a set of *positions*. A *virtual processor* operates on parallel data at each position. The underlying computer multiplexes a set of physical processors among the virtual processors. Scalar data remains global to all virtual processors. This model of programming is also known as SPMD, for Single Program, Multiple Data, a more loosely synchronized software implementation of the SIMD (Single Instruction, Multiple Data) model.

Each parallel variable in C* has a *shape*, which describes the logical structure of positions. At any point in the program, a *current shape* is in force. *Elemental* parallel operations operate elementwise on data of the current shape. A `with` statement selects the current shape, which is denoted 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—those whose virtual processors execute parallel operations. A `where` statement narrows the context, like a parallel `if` statement, by selecting as active a subset of the active positions within the shape. An `everywhere` statement makes all positions active. Exiting a `where` or `everywhere` statement restores the context in force before the statement. Functions inherit the current context and the current shape from their caller.

Parallel *communication* transfers parallel data among the virtual processors. There are several forms of parallel communication. *Reductions* combine elements of a parallel variable

```
void  filter(double:current *envelope); /* external computation */

void harmonize() {
  const N = 1 << 30;                    /* N == 2 ** 30 */
  outofcore shape [N]series;            /* series of terms */
  with (series) {                       /* set current shape */
    long int k:current = pcoord(0);     /* index array */
    where (k > 0) {                     /* avoid computing 1/0 */
      double harm:current = 1.0 / k;    /* 1/k (0 < k < N) */
      filter(&harm);                    /* process */
      return += harm;                   /* sum result */
    }
  }
}
```

Figure 1: A sample ViC* program to compute in parallel the terms $1/k$ for $0 < k < 2^{30}$ out of core, call an external function, and return their sum.

into a scalar result, for example summing the elements. *Left indexing* a parallel variable stores or extracts a scalar value at a single position. For example, if `a` is a parallel integer, `[5]a` denotes the fifth position in `a`. *Parallel left indexing* addresses data in a set of virtual processors. If `b` is also a parallel variable, `[b]a` denotes `[[`i`]b]a` in each position i. Virtual processors executing a *get operation* in an expression fetch data from other virtual processors, and 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.

C* is based on a distributed-memory model of parallel data, with data spread across separate address spaces. In contrast to C arrays, the address of a position in a C* parallel variable is not denotable, and hence individual positions of parallel variables cannot be addressed with pointers. All communication among positions takes place through the explicit communication operations. ViC* implements virtual memory for parallel data by exploiting the distributed-memory model to place out-of-core data on disks and load it into memory as needed.

An Example

The sample ViC* program in Figure 1 illustrates the use of `outofcore` data. The example computes a partial harmonic series for the first $N-1$ terms, or $(1/k)$ for $k = 1, 2, \ldots, N-1$, passes its address to an external function, and sums the result. Although the example does not necessarily demonstrate the most efficient means of computing this series, it illustrates a number of issues and optimizations for processing out-of-core data.

The function `harmonize()` declares an out-of-core shape `series` with 2^{30} positions stored on disk. The `with` statement establishes `series` as the current shape. Since `series` is a

new shape, the associated context is `everywhere`.

An index variable, `k`, is initialized with the `pcoord()` parallel intrinsic function. The call to `pcoord()` returns the index set along a dimension (in this case, 0) of the current shape. Here it returns `0` in position 0, `1` in position 1, through $2^{30} - 1$ in the last position. The `where` statement narrows the context to positions 1 through $2^{30} - 1$. The parallel variable `harm` is assigned the reciprocals of `k`, i.e., $1/i$ in each position i. Having narrowed the context avoids a division by zero in this expression.

The example calls an external function, `filter()`, with a pointer to `harm`. Function `filter()` also inherits the current shape and context. Such an external call limits the scope of optimizations as described below. On return, a sum-reduction, denoted by the overloaded `+=` operator in C*, returns the sum of the resulting elements of `harm`.

4 Loop Transformations

In this section we discuss transformations of parallel loops—loop fusion, rematerialization and dead store elimination, and scalarization—that improve their performance on out-of-core data. We illustrate their effect on the example from Figure 1 and compare the page I/O counts for parallel data.

Figure 2 shows a C code schema that implements the statements inside the `with` statement of function `harmonize()`. The `PASS` construct describes a parallel loop over each position. Within this loop, the `ELT` construct selects the element at the current position. Each `if` statement restricts execution to the active positions.

The global pointer `CONTEXT` points to the current context; a null pointer indicates an `everywhere` context. The second loop computes the `where` context into an auxiliary parallel variable, `where_1`, which is used in subsequent loops. The global `CONTEXT` is stacked in `context_0` and popped at the end of the `where` statement; since functions inherit context when called, `filter()` will also reference `CONTEXT`.

Several of the C* constructs invoke additional runtime support. `PCOORD()` references the current position for the `pcoord` intrinsic. In a multiprocessor system, `sum_reduce()` combines the partial sums, accumulated in `red_2`, from each processor and distributes the result to all processors.

Consider the behavior of the C program scheme in Figure 2 with large data sets in a traditional sequential demand-paged environment. Each `PASS` loop will typically page in all its parallel data operands and write back all parallel results. Consider a single-processor system, such as a DEC Alpha running OSF/1, where an 8-KB page holds 1K `double`s or `long`s, or 8K booleans (implemented as bytes). We consider only paging due to parallel data, and we assume that the current shape is large enough that no pages are still in memory by the time they are referenced in a subsequent pass. Table 1 shows the expected number of page transfers for parallel variables in each each loop. The total page traffic for this version of the program is 5504K pages. Page prefetching can be used to reduce the latency but does not affect this total I/O count.

```
bool (*context_0): current = CONTEXT;       /* current context */
bool where_1: current;                      /* context value */
double red_2 = 0;                           /* sum accumulator */

PASS {                                      /* LOOP 1 */
  if (!context_0 || ELT(*context_0))        /* inherited context */
    ELT(k) = PCOORD(0);                     /* set k */
}

PASS {                                      /* LOOP 2 */
  ELT(where_1) = (!context_0 || ELT(*context_0)) && (ELT(k) > 0);
}

CONTEXT = &where_1;                         /* push context */

PASS {                                      /* LOOP 3 */
  if (ELT(where_1))                         /* in new context */
    ELT(harm) = 1.0 / ELT(k);               /* set envelope */
}

filter(&harm);                              /* call */

PASS {                                      /* LOOP 4 */
  if (ELT(where_1))                         /* in new context */
    red_2 += ELT(harm);                     /* sum envelope */
}

red_2 = sum_reduce(red_2);                  /* combine sum */

CONTEXT = context_0;                        /* pop context */

return red_2;
```

Figure 2: C schema for loops in `harmonize()` in Figure 1. The `PASS` construct iterates over all positions, and `ELT` evaluates the current position. The variable `context_0` stacks the initial context. Temporary variables `where_1` and `red_2` store the context and summation, respectively.

loop	`k`	`where_1`	`harm`	*total*
1	1024K			1024K
2	1024K	128K		1152K
3	1024K	128K	1024K	2176K
4		128K	1024K	1152K
program	3072K	384K	2048K	5504K

Table 1: Page I/O counts for the program schema in Figure 2 with 8-KB pages.

```
bool (*context_0): current = CONTEXT;       /* inherited context */
bool where_1: current;                      /* context value */
double red_2 = 0;                           /* sum accumulator */

PASS {                                      /* LOOP 1' */
  if (!context_0 || ELT(*context_0))        /* inherited context */
    ELT(k) = PCOORD(0);                     /* set k */
  ELT(where_1) = (!context_0 || ELT(*context_0)) && (ELT(k) > 0);
  if (ELT(where_1)) {                       /* in new context */
    ELT(harm) = 1.0 / ELT(k);               /* set envelope */
  }
}

CONTEXT = &where_1;                         /* push context */

filter(&harm);                              /* call */

PASS {                                      /* LOOP 2' */
  if (ELT(where_1))                         /* in new context */
    red_2 += ELT(harm);                     /* sum envelope */
}

red_2 = sum_reduce(red_2);                  /* combine sum */

CONTEXT = context_0;                        /* pop context */

return red_2;
```

Figure 3: C schema for `harmonize()` in Figure 1 with the first four loops of Figure 2 fused into a single loop.

Loop Fusion

Loop fusion is a transformation which combines adjacent loops with similar bounds where data dependencies in the loop bodies permit. Loop fusion reduces loop overhead and, more significantly for out-of-core data, improves data locality when the loops access the same data.

Figure 2 represents a straightforward translation of the ViC* program in Figure 1, in which every parallel operation becomes a loop. A more sophisticated C* compiler [LH92] would fuse these loops, as illustrated in Figure 3. The first three loops of Figure 2 have been fused into a single `PASS`. The remaining loop cannot be fused because of data dependencies: the sum depends on any modifications to elements of `harm` in the call to `filter()`.

In Figure 3, parallel data is reused within the fused loop. All references to `k` are in the first loop, and so this data is traversed exactly once. Variable `where_1` is traversed twice. Table 2 shows the resulting page I/O counts.

loop	`k`	`where_1`	`harm`	*total*
$1'$	1024K	128K	1024K	2176K
$2'$		128K	1024K	1152K
program	1024K	256K	2048K	3328K

Table 2: Page I/O counts for the program schema in Figure 3 after loop fusion.

Rematerialization

Rematerialization [BCT92] is a transformation that recomputes a variable instead of using its stored value—the inverse of common subexpression elimination. In other compilers, rematerialization typically reduces the number of memory accesses. ViC* uses rematerialization to reduce the number of out-of-core reads at the cost of additional computation. In most current architectures, the cost of disk I/O greatly outweighs the cost of computation. Figure 4 shows the effect of rematerialization on the example.

Rematerialization can be applied when a variable's value is computable from in-core data. Parallel values computed from scalars and the `pcoord()` function are suitable for rematerialization, as are variables computed from other values already available in a loop. A context which may be `everywhere`, such as the one established by the `with` statement, is also a good candidate for rematerialization, since it may require no data. In the second loop of Figure 4 the context value, `where_1` is rematerialized from the initial context and the expressions `pcoord(0)` and `k > 0`. The resulting page I/O counts are shown in Table 3. Total page traffic is reduced to 3200K pages.

Dead Store Elimination

Dead store elimination is applied after rematerialization has eliminated references to stored parallel variables. Because the local parallel variable `k` is rematerialized in the second loop, its value is never read from disk. Consequently, the assignment to `k` is *dead*, and so its computed vales need not be written to disk. On the other hand, the current context in `where_1` is implicitly passed to `to_grid()`, so it must be written.

Determining that a parallel assignment is dead requires dataflow analysis of variables shared between loops. Dead store elimination does not alter the loop structure, but it eliminates the need to write back the out-of-core data. The resulting page I/O counts are shown in Table 4. The total page I/O count has been reduced to less than half the original count.

Scalarization

Scalarization replaces parallel variables with scalars. It applies to intermediate parallel variables that are used only within a loop and neither read nor written to storage. Scalarization reduces the overhead of selecting a position as well as the demand for memory in a loop.

```
bool (*context_0): current = CONTEXT;       /* inherited context */
bool where_1: current;                      /* context value */
double red_2 = 0;                           /* sum accumulator */

PASS {                                      /* LOOP 1'' */
  if (!context_0 || ELT(*context_0))        /* inherited context */
    ELT(k) = PCOORD(0);                     /* set k */
  ELT(where_1) = (!context_0 || ELT(*context_0)) && (ELT(k) > 0);
  if (ELT(where_1)) {                       /* in new context */
    ELT(harm) = 1.0 / ELT(k);               /* set envelope */
  }
}

CONTEXT = &where_1;                         /* push context */

filter(&harm);                              /* call */

PASS {                                      /* LOOP 2'' */
  if (!context_0 || ELT(*context_0))        /* inherited context */
    ELT(k) = PCOORD(0);                     /* rematerialize new context */
  ELT(where_1) = (!context_0 || ELT(*context_0)) && (ELT(k) > 0);
  if (ELT(where_1)) {
    red_2 += ELT(harm);                     /* sum envelope */
  }
}

red_2 = sum_reduce(red_2);                  /* combine sum */

CONTEXT = context_0;                        /* pop context */

return red_2;
```

Figure 4: Rematerialization. The values of `k` and `where_1` are recomputed rather than being read from storage.

Scalarization is applied after rematerialization has eliminated reads and dead store elimination has eliminated writes. In Figure 5, this transformation is applied to the local variable `k`. In the second `PASS`, the current context is also replaced with a scalar. Only `harm` and `where_1` in the first loop and `harm` in the second loop remain as out-of-core variables. These are exactly the variables visible to the function `filter()`.

Scalarization requires dataflow analysis of variables shared within a loop. Although scalarization does not affect the page I/O requirements, it improves in-core access within the `PASS` loops, an important consideration for in-core shapes.

loop	`k`	`where_1`	`harm`	*total*
$1''$	1024K	128K	1024K	2176K
$2''$			1024K	1024K
program	1024K	128K	2048K	3200K

Table 3: Page I/O counts for the program schema in Figure 4 after rematerialization.

loop	`k`	`where_1`	`harm`	*total*
$1'''$		128K	1024K	1152K
$2'''$			1024K	1024K
program		128K	1024K	2176K

Table 4: Page I/O counts for the program schema in Figure 4 after dead store elimination.

Summary

For this example, the loop transformations described above reduce page I/O counts by 60%. Experienced programmers often apply such optimizations, particularly when I/O is explicit. As can be seen by comparing Figures 1 and 5, the transformations tend to obscure the program structure. Adding I/O calls results in a program more like Figure 9. This sequence of transformations is the sort of programming process that is better left to a compiler such as ViC*.

5 Data Layout

In this section, we consider the structure of C* memory and its mapping onto a linear in-core address space. C* data parallelism is orthogonal to the conventional C address space. For in-core access, this two-dimensional memory can be mapped onto a linear address space in one of two layouts. To illustrate these memory layouts, Figure 6 depicts a parallel structure `p` declared with shape `s`. The parallel elements are selected by left indexing positions `0` through `999`. The structure fields `x` and `y` are located sequentially in a conventional C address space. Each field of `p`, i.e. `p.x` and `p.y`, is itself a parallel `int`, all of whose parallel elements share the same C address.

Struct-major layout (column-major in Figure 6) keeps each struct in contiguous memory, as shown in Figure 7. This layout conforms to the C layout for an array of structures. Operations on parallel struct elements simply iterate operations on scalar structs. Each structure field, however, is not stored in contiguous memory, but is strided through the structure array. A field address, such as `&p.x`, must include this stride information. An operation on a parallel field, such as `p.x++`, is implemented with a loop that extracts this field from each structure element. For out-of-core parallel data this extraction may require

```
bool (*context_0): current = CONTEXT;       /* inherited context */
bool where_1: current;                      /* context value */
double red_2 = 0;                           /* sum accumulator */

PASS {                                      /* LOOP 1''' */
  long k_3;                                 /* scalar ELT(k) */
  if (!context_0 || ELT(*context_0))        /* inherited context */
    k_3 = PCOORD(0);                        /* set k */
  ELT(where_1) = (!context_0 || ELT(*context_0)) && (k_3 > 0);
  if (ELT(where_1)) {                       /* in new context */
    ELT(harm) = 1.0 / k_3;                  /* set envelope */
  }
}

CONTEXT = &where_1;                         /* push context */

filter(&harm);                              /* call */

PASS {                                      /* LOOP 2''' */
  long k_4;                                 /* scalar ELT(k) */
  bool where_5;                             /* scalar ELT(where_1) */
  if (!context_0 || ELT(*context_0))        /* inherited context */
    k_4 = PCOORD(0);                        /* rematerialize new context */
  where_5 = (!context_0 || ELT(*context_0)) && (k_4 > 0);
  if (where_5) {
    red_2 += ELT(harm);                     /* sum envelope */
  }
}

red_2 = sum_reduce(red_2);                  /* combine sum */

CONTEXT = context_0;                        /* pop context */

return red_2;
```

Figure 5: Scalarization. Parallel variable `k` and, in the second loop, the current context are replaced by scalar temporaries.

```
shape [1000] s;
struct { int x,y; } p: s;
```

[0]p.x	[1]p.x	[2]p.x	...	[998]p.x	[999]p.x
[0]p.y	[1]p.y	[2]p.y	...	[998]p.y	[999]p.y

Figure 6: Parallel memory. The structure `p` is a parallel structure with shape `s`, having 1000 positions. The structure member offsets a conventional C address, shown vertically. The left index selects a position 0,1, ..., 999 in `p`, shown horizontally.

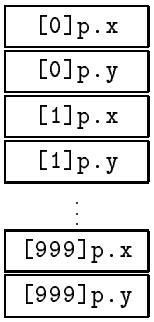

Figure 7: Struct-major parallel memory layout. Struct-major layout for Figure 6 keeps the fields of each position together.

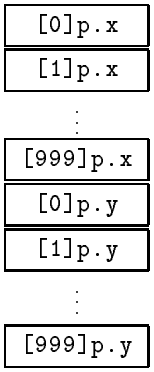

Figure 8: Field-major parallel memory layout. Field-major layout for Figure 6 keeps each field together as if it were a separate parallel variable.

transferring the entire structure to and from disk. In-core data also suffers reduced data locality in cache.

Field-major layout (row-major in Figure 6) keeps each field in contiguous memory, as shown in Figure 8. This layout conforms to the C layout for a structure of arrays. A field address, such as `&p.x`, addresses a contiguous range of memory. An operation on a parallel field, such as `p.x++`, operates on contiguous data. Operations on parallel structs must be split into structure field operations. A structure assignment such as `q = p`, where `q` is a compatible structure, is implemented as a series of parallel assignments

```
q.x = p.x;
q.y = p.y;
```

ViC* uses field-major layout to improve locality for field accesses. Field-major layout also simplifies out-of-core data movement. It limits the size of a parallel data element to the largest primitive data type, here a `double`. Each disk block can hold an integral number of elements, and so no element needs to be split across blocks. ViC* also uses an integral

number of disk blocks for each parallel variable, so that each block holds elements of a single variable. This layout reduces the need to do read-modify-write operations when writing parallel data.

Field-major layout has drawbacks as well. The underlying ViC* I/O library operates on logical blocks which are a multiple of the disk block size, each containing the same number of parallel elements. A logical block of `double`s is eight times the size of a logical block of bytes. ViC* I/O calls operate on a number of elements instead of a byte count. In the runtime interface described in Section 6, all out-of-core data transfers in a loop operate on the same number of elements. Because the logical block size must be known, with field-major layout we cannot support the `palloc()` function, which allocates an untyped block of parallel storage.

6 Runtime Support

In this section, we describe the ViC* runtime I/O interface for parallel data loops. After the loops have been restructured, the ViC* compiler inserts I/O calls and expands parallel variable references.

Figure 9 shows the final C code of `harmonize()` after applying all the transformations in Section 4. Each `PASS` loop is expanded into an outer *sectioning* loop and an inner in-core vector loop. References to parallel variables are replaced by references to in-core strips. Other calls set up I/O and manage loop iteration.

Each `PASS` loop begins with a call to `ACCESS_DATA()` for each parallel variable used in the loop. This function establishes a virtual-memory mapping for out-of-core data and describes the access as read-only (`'R'`), write-only (`'W'`), modify (read and write, `'M'`), or read-context (`'C'`). The `ACCESS_DATA()` function detects aliasing among parallel data references and combines the references. It also verifies that the parallel data has the current shape.

At first glance, it might appear that reading a context is no different than reading any other parallel `bool`. It turns out that there are two reasons to treat the context specially. First, it provides a simple programming check. Recall that the pointer to a context may be null, which would indicate an `everywhere` context. In such a case, no read would occur. For all other out-of-core parallel data, the pointer must be non-null; an access type other than read-context with a null data pointer triggers a runtime error. (The origin of such an error would be the ViC* compiler rather than the programmer.) Second, treating the context specially enables a runtime optimization. Again supposing that the context pointer is null, consider what happens when we perform write-only access on some other out-of-core parallel variable `a`. Because the context is `everywhere`, there is no need to first read `a`; every position will be written. On the other hand, when the context is not known to be `everywhere`, we must first read `a` in order to maintain values in the inactive positions. When `ACCESS_DATA()` finds that the context is `everywhere`, it can automatically optimize access to write-only data within the loop.

```
bool (*context_0): current = CONTEXT;                 /* inherited context */
bool where_1: current;                                /* context value */
double red_2 = 0;                                     /* sum accumulator */

ACCESS_DATA(context_0,'C')                            /* loop over context */
ACCESS_DATA(&where_1,'W')                             /* loop over where_1 */
ACCESS_DATA(&harm,'W')                                /* loop over harm */
while (ITERATE_STRIPS()) {                            /* LOOP 1'''' */
  int vp_10;                                          /* VP index */
  bool *strip_11 = (bool*)INCORE_STRIP(context_0);    /* in-core data */
  bool *strip_12 = (bool*)INCORE_STRIP(&where_1);
  double *strip_13 = (double*)INCORE_STRIP(&harm);
  for (vp_10 = COUNT_STRIP(); 0<=--vp_10; ITERATE_PCOORD()) {
    long k_3;                                         /* scalar ELT(k) */
    if (!context_0 || strip_11[vp_10])                /* inherited context */
      k_3 = PCOORD(0);                                /* set k */
    strip_12[vp_10] = (!context_0 || strip_11[vp_10]) && (k_3 > 0);
    if (strip_12[vp_10]) {                            /* in new context */
      strip_13[vp_10] = 1.0 / k_3;                    /* set envelope */
    }
  }
}

CONTEXT = &where_1;                                   /* push context */

filter(&harm);                                        /* call */

ACCESS_DATA(context_0,'C')                            /* loop over context */
ACCESS_DATA(&harm,'R')                                /* loop over harm */
while (ITERATE_STRIPS()) {                            /* LOOP 2'''' */
  int vp_13;                                          /* VP index */
  bool *strip_14 = (bool*)INCORE_STRIP(context_0);    /* in-core data */
  double *strip_15 = (double*)INCORE_STRIP(&harm);
  for (vp_13 = COUNT_STRIP(); 0<=--vp_13; ITERATE_PCOORD()) {
    long k_4;                                         /* scalar ELT(k) */
    bool where_5;                                     /* scalar ELT(where_1) */
    if (!context_0 || strip_14[vp_13])
      k_4 = PCOORD(0);
    where_5 = (!context_0 || strip_14[vp_13]) && (k_4 > 0);
    if (where_5) {
      red_2 += strip_15[vp_13];                       /* sum envelope */
    }
  }
}

red_2 = sum_reduce(red_2);                            /* combine sum */

CONTEXT = context_0;                                  /* pop context */

return red_2;
```

Figure 9: Expanded loop code. PASS constructs have been expanded into inner in-core and outer sectioning loops. Parallel data is prefetched and accessed through in-core strip pointers.

Within the `PASS`, all data movement is managed by `ITERATE_STRIPS()`, which returns a true value as long as there are additional parallel data to process. For in-core shapes no I/O is required, and `ITERATE_STRIPS()` returns true only once. For out-of-core shapes, `ITERATE_STRIPS()` also manages I/O, prefetching the next strip and writing back previous strips. I/O is striped across disks in a parallel disk system. Asynchronous I/O is overlapped with in-core computation.

Once data is in-core, `INCORE_STRIP()` locates the current in-core strip of each parallel variable. All these strips have the same number of in-core positions, as returned by `COUNT_STRIP()`, regardless of the element size. Thus, for example, there are as many in-core elements of `harm` as of `context_0` in the second `PASS` loop of Figure 9, although they occupy eight times as many pages.

The inner loop processes in-core data, iterating a virtual processor index through the in-core positions. Within the inner loop, all parallel data references are replaced by strip array references, indexed by the virtual processor number. If necessary, each iteration calls `ITERATE_PCOORD()` to set the next next value of `PCOORD(0)`. The resulting code structure encourages optimization by the final C compiler.

7 Performance

We measured performance of the loops in Figure 9, varying the number of positions in the shape `series` and characteristics of parallel data access. The test system is a DEC 2100 server with two 175-MHz Alpha processors, eight disks, and 320 MB of main memory. For each position in the shape `series` the first loop writes 9 bytes of data and the second loop reads 8 bytes. For this example, the main memory capacity is a little over 35 million positions. Out-of-core data is accessed through the file system by way of a 64 MB in-core buffer pool.

Figure 10 shows the average time per position for problem sizes ranging from 1 million to 100 million positions. An in-core shape relies on demand paging to access the data. Out-of-core shapes manage I/O through the file system. The standard ViC* implementation uses asynchronous I/O, but a synchronous implementation reduces the memory buffer requirements. The number of parallel disks varied among 1, 2, 4, and 8.

In-core data access with demand paging outperforms out-of-core data access at small problem sizes, but it degrades rapidly at large problem sizes. Below 30 million positions, there is little paging, and the time to process each position averages around one microsecond. As the problem size approaches the main memory size, demand paging degrades performance sharply. For problem sizes above 50 million positions, demand paging adds about 5 microseconds to the processing time for each position. An address space of 1 GB limits in-core problem sizes to 100 million positions.

Although out-of-core data access shows higher overhead costs at small problem sizes, it maintains performance at large problem sizes. At small problem sizes, fixed-cost loop setup operations dominate. The system buffer cache makes a copy of out-of-core data as it is read or written, thus adding a constant to the time for each position. The buffer cache is effective

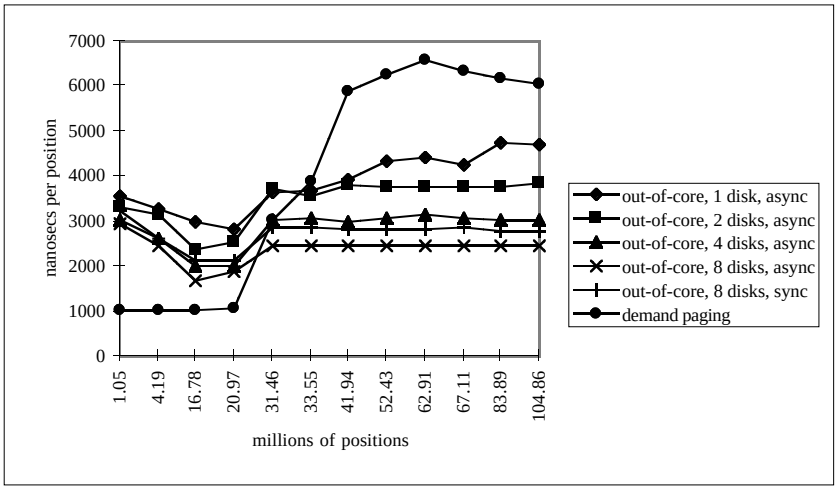

Figure 10: Performance of the loops in Figure 9 with in-core and out-of-core data implementations.

in reducing I/O below 30 million positions, the region where demand paging is also effective, but at a higher cost. As the problem size becomes greater than the size of main memory, out-of-core data-access times remain stable. At 100 million positions, out-of-core data access across eight disks takes between 2 and 3 microseconds per position.

Even with a single disk, the ViC* out-of-core code outperforms demand paging at large problem sizes. Increasing the number of parallel disks further reduces the access time. Synchronous I/O across eight disks reduces performance somewhat by removing overlap between computation and I/O, but the ViC* code continues to overlap I/O among parallel disks. The buffer pool is large enough that the increased buffer sizes available for synchronous I/O provide little benefit.

8 Conclusion

We have described an implementation of virtual memory for out-of-core data-parallel programming with a parallel disk system. ViC* divides responsibility for memory management between the programmer, who declares `outofcore` shapes, the compiler, which restructures parallel operations, and the runtime system, which manages I/O data transfers and buffering. This approach differs from conventional demand-paged virtual memory, which operates at the level of instructions and memory references.

By using explicit runtime I/O interfaces, the ViC* compiler is able to manage parallel virtual memory at a higher level, coordinating the out-of-core data transfers across an entire loop. The compiler also emits calls to a runtime library that invokes efficient parallel disk algorithms for out-of-core communication functions. In both ways, the ViC* system significantly reduces disk-access costs for out-of-core data.

Acknowledgments

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

[AP94] Alok Aggarwal and C. Greg Plaxton. Optimal parallel sorting in multi-level storage. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 659–668, January 1994.

[Arg95] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *4th International Workshop on Algorithms and Data Structures (WADS)*, pages 334–345, August 1995.

[AVV95] Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in geographic information systems. In Paul Spirakis, editor, *Proceedings of the Third Annual European Symposium on Algorithms (ESA '95)*, volume 979 of *Lecture Notes in Computer Science*, pages 295–310. Springer-Verlag, September 1995.

[BCT92] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 311–321, June 1992.

[BGV97] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4–5):601–631, June 1997.

[CC94] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dartmouth College Department of Computer Science, November 1994.

[CGG+95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, January 1995.

[CH97] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the Parallel Disk Model with the ViC* implementation. *Parallel Computing*, 23(4–5):571–600, June 1997.

[CN96] Thomas H. Cormen and David M. Nicol. Performing out-of-core FFTs on parallel disk systems. Technical Report PCS-TR96-294, Dartmouth College Department of Computer Science, August 1996. To appear in *Parallel Computing*.

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

[Cor97] Thomas H. Cormen. Determining an out-of-core FFT decomposition strategy for parallel disks by dynamic programming. Technical Report PCS-TR97-322, Dartmouth College Department of Computer Science, July 1997. To appear in [IMA96].

[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. Preliminary version appeared in *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*. Revised version to appear in *SIAM Journal on Computing*.

[CWN97] Thomas H. Cormen, Jake Wegmann, and David M. Nicol. Multiprocessor out-of-core FFTs with distributed memory and parallel disks. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS '97)*, pages 68–78, November 1997. Also Dartmouth College Computer Science Technical Report PCS-TR97-303.

[dDEF+97] B. D. de Dinechin, G. Elsesser, G. Fischer, B.H. Johnson, T. MacDonald, R. W. Numrich, and Jon L. Steidel. Definition of the F^{--} extension to Fortran 90. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, August 1997. To appear.

[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.

[IMA96] *Proceedings of the Workshop on Algorithms for Parallel Machines*, 1996–97 Special Year on Mathematics of High Performance Computing, Institute for Mathematics and Its Applications, University of Minnesota, Minneapolis, September 1996.

[LH92] Anthony J. Lapadula and Kathleen P. Herold. A retargetable C* compiler and run-time library for mesh-connected MIMD multicomputers. Technical

Report TR 92-15, University of New Hampshire, 1992. Revised by Phil Hatcher, October 1993.

[NV93] Mark H. Nodine and Jeffrey Scott Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, June 1993.

[NV95] Mark H. Nodine and Jeffrey Scott Vitter. Greed sort: Optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, July 1995.

[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.

[Wis96] Leonard F. Wisniewski. *Efficient Design and Implementation of Permutation Algorithms on the Memory Hierarchy.* PhD thesis, Dartmouth College Department of Computer Science, March 1996.