Piecewise Execution of Nested Data-Parallel Programs

Daniel W. Palmer, Jan F. Prins, Siddhartha Chatterjee, and Rickard E. Faith

Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27599-3175
{palmerd,prins,sc,faith}@cs.unc.edu

Abstract. The technique of flattening nested data parallelism combines all the independent operations in nested apply-to-all constructs and generates large amounts of potential parallelism for both regular and irregular expressions. However, the resulting data-parallel programs can have enormous memory requirements, limiting their utility. In this paper, we present *piecewise execution*, an automatic method of partially serializing data-parallel programs so that they achieve maximum parallelism within storage limitations. By computing large intermediate sequences in pieces, our approach requires asymptotically less memory to perform the same amount of work. By using characteristics of the underlying parallel architecture to drive the computation size, we retain effective use of a parallel machine at each step. This dramatically expands the class of nested data-parallel programs that can be executed using the flattening technique. With the addition of piecewise I/O operations, these techniques can be applied to generate out-of-core execution on large datasets.

1 Introduction

1.1 Flattening nested data parallelism

Nested data parallelism is a powerful paradigm for expressing concurrent execution, especially irregular and dynamic computations. Unlike flat data-parallel languages such as C* [11], High Performance Fortran [9], and APL [12], nested data-parallel languages allow arbitrary functions to appear in apply-to-all constructs and provide nestable, non-rectangular aggregates. The expressive benefits of nested data parallelism were long ago recognized by high-level languages such as SETL [19], FP [2], and APL2, but practical parallel execution of such expressions was not achieved until Blelloch and Sabot introduced the flattening technique [4]. Flattening combines all the independent operations in nested apply-to-all constructs into large data-parallel operations. Both NESL [5] and Proteus [13] are high-level, nested data-parallel languages that use this technique to provide architecture-independence by implementing the data-parallel operations with portable vector operations [6].

1.2 Excessive memory requirements of flattened programs

The flattening technique fully parallelizes every apply-to-all construct, providing large amounts of fine-grained potential parallelism, but introduces temporaries whose sizes are proportional to the potential parallelism. The generality of the apply-to-all construct can easily lead to programs that have enormous potential parallelism, and hence, excessive memory requirements. Blelloch and Narlikar [3] encountered these large temporaries while comparing two algorithms for n-body simulations. They resolved the problem by manually serializing portions of their NESL code reducing the program's memory requirements so it could execute.

The following example illustrates that flattening a nested data-parallel expression can generate code with large temporary values. Consider a problem related to the n-body computation: finding the largest force between any two particles in a sequence of n particles. We express this computation in Proteus using a nesting of two data-parallel $iterators$.

```
max./[i in [1..n]:
          max./[j in [1..i-1]: force(S[i],S[j])]]        (1)
```

Here `force(`p`,`q`)` yields the magnitude of the force between particles p and q. The inner iterator specifies a sequence of independent applications of `force` and the outer iterator specifies a sequence of independent reductions. The dependence of the inner iterator variable, `j`, on the outer iterator variable, `i`, ensures that we only compute the force between any two particles once. The dependency also generates an irregular collection of arguments to `force`. Consequently, the computation requires an irregular data aggregate which flat data-parallel languages do not linguistically support. Flattening (1) combines the n separate inner invocations of `max./` into a single, larger data-parallel maximum reduction and also combines the $\frac{n(n+1)}{2}$ nested invocations of `force` into a single invocation of the data-parallel version of the function. As a consequence, $O(n^2)$ applications of `force` are evaluated simultaneously, requiring $O(n^2)$ storage. Clearly, we could sequentially evaluate the expression using only $O(n)$ space. Note that flattening can handle arbitrary, user-defined functions in place of `force` and `max./`. In this paper, we show how to partially serialize flattened programs to reduce their memory requirements while still retaining sufficient parallelism to fully utilize the resources of a targeted architecture.

1.3 Organization of paper

The remainder of this paper is organized as follows. In Section 2, we examine two approaches to partially serializing data-parallel operations: outer iterator serialization and piecewise execution. We further explore piecewise execution in Section 3 and present an implementation of interpreted piecewise execution. In Section 4, we identify some key issues in compiling piecewise execution programs. We present some preliminary performance results of piecewise execution in Section 5. Then, in Section 6, we identify some limitations of piecewise execution and finally, in Section 7, we discuss related work, report the status of our current system, an present our conclusions.

2 Fixed Memory Execution of Flattened Programs

2.1 Partially serialized parallelism

By combining all the operations in nested iterators, flattening exposes all the available parallelism and generates a data-parallel program that operates on sequences, potentially very large ones. A hypothetical parallel machine with an arbitrary number of processors could take advantage of all this parallelism by executing data-parallel operations in a single step (see Fig. 1a). Existing, fixed-processor parallel machines could attempt to execute the data-parallel operations with virtual processors. This approach works well for the large class of flattened programs whose memory requirements do not exceed the resources of a targeted machine. However, in general, simulating n virtual processors requires $O(n)$ memory. To realistically execute a flattened program on existing parallel machines, we must partially serialize the program to reduce its memory requirements.

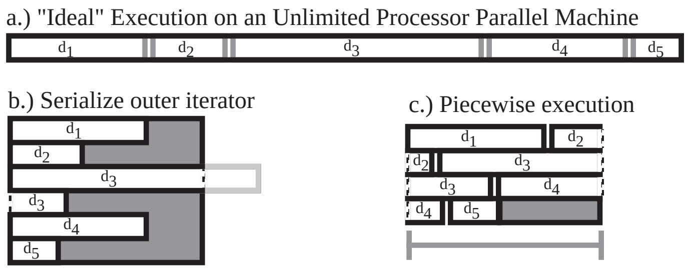
a.) "Ideal" Execution on an Unlimited Processor Parallel Machine
d_1
d_2
d_3
d_4
d_5
b.) Serialize outer iterator
c.) Piecewise execution

Fig. 1. Approaches to Executing Potential Parallelism

Instead of generating monolithic data-parallel operations from nested iterator expressions, we could transform outer iterators into loop structures and inner iterators into smaller data-parallel operations. This approach serializes regular parallelism well, because each loop iteration will execute the same amount of work, yielding good load balancing. However, for irregular parallel expressions, like the n-body force computation, serializing the outer iterations yields large variations in work and wastes computational resources on undersized sequences (see Fig. 1b).

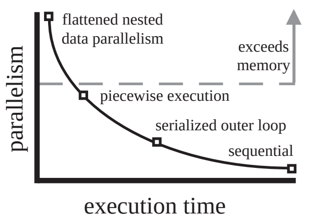

Fig. 2. Comparison of Execution Methods

To achieve load-balanced execution, we must always serialize a computation into *equal* sized pieces. To execute on many different platforms, we must select the size of these pieces based on the characteristics of the target architecture, not simply on the characteristics of the computation. Since it is much easier to serialize portions of exposed parallelism than it is to extract a specified amount of parallelism from nested iterators, we will not alter the the flattening technique. Instead, we introduce a new technique that partitions large data-parallel operations into uniformly-sized pieces (see Fig. 1c). This *piecewise execution* allows a program to retain sufficient parallelism for good performance and satisfies the memory resource restrictions shown in Fig. 2.

2.2 Piecewise execution of flattened data-parallel programs

Fig. 3 illustrates an overview of our implementation of flattening and transformation of high-level nested data-parallel programs into executable vector operations.

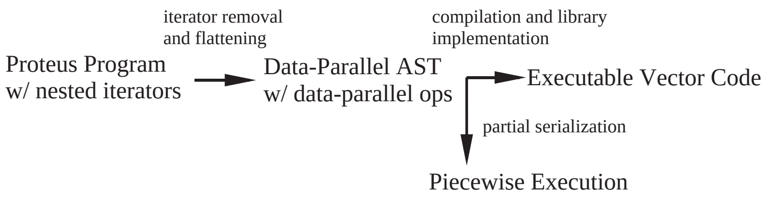
iterator removal and flattening
compilation and library implementation
Proteus Program w/ nested iterators
Data-Parallel AST w/ data-parallel ops
Executable Vector Code
partial serialization
Piecewise Execution

Fig. 3. Generating Executable Code from Nested Data-Parallel Programs

To present a concrete example of this process and to illustrate piecewise execution, we consider computing $n!$ using multiplication reduction on the sequence $[1,\ldots,n]$. We can write this in Proteus as `*./[i in [1..n]:i]`. We then successively apply iterator-removing transformation rules [18] [15], yielding equivalent data-parallel operations.

```
*./[i in [1..n]:i]
= mult_reduce([i in [1..n]:i])
= mult_reduce([1..n])
= mult_reduce(range1(n))
```

These operations are part of the Data-Parallel Library (DPL) [14], a collection of routines that supports nested sequences as primitive objects and provides data-parallel execution of nested sequence operations. We then compile the functional expression into an imperative, single-assignment form with explicit temporary variables.

```
T = range1(n);
r = mult_reduce(T);
```

The function `range1(n)` generates an enumerated sequence of integers from 1 to n and `mult_reduce(T)` computes the product of the values in T. To compute several factorials simultaneously, we put the values in a sequence $D = [d_1,\ldots,d_n]$, and use `[i in D: */.[j in [1..i]:j]]` to yield $[d_1!, \ldots, d_n!]$. Although this algorithm is not work efficient, it provides a useful example for illustrating piecewise execution. Evaluated in this manner the Proteus code specifies a data-dependent, irregular parallel operation. Flattening yields the following data-parallel operations.

```
  [i in D: mult_reduce([j in [1..i]:j])]
= mult_reduce¹([i in D:[j in [1..i]:j]])
= mult_reduce¹([i in D:[1..i]])
= mult_reduce¹([i in D:range1(i)])
= mult_reduce¹(range1¹(D))
```

For any function f, we use f^1 to designate a data-parallel function that applies f to all elements of a sequence in parallel. In this example, `range`^{1} computes multiple enumerations in parallel and `mult_reduce`^{1} computes many sequence products simultaneously. As before, we rewrite the functional expression in a single-assignment form with explicit temporary values.

```
T = range1¹(D);
R = mult_reduce¹(T);
```

The data-parallel version of `range1` generates results whose storage can greatly exceed that of the inputs. Conversely, `mult_reduce`^{1} produces results whose storage requirements can be far less than that of its inputs. Fig. 4 shows the data-flow graph for the data-parallel factorial program using these operations. The trapezoid-shaped nodes indicate the relative size relation between the inputs and outputs of an operation.

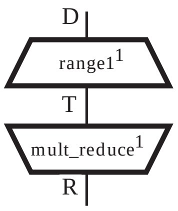

Fig. 4. Simple Data-Flow Graph

As illustrated in the maximum force example, the large temporary, T, can exhaust memory. This is of particular concern for two reasons. First, the large memory requirements are not inherent in the original program, but are introduced by the flattening process. Second, whether a program will exceed memory or not depends on the particular target architecture and on the particular problem size, both of which are determined at runtime. To resolve these problems, we must generate partially serialized code from the data-parallel abstract syntax tree. We can express the functionality of a *piecewise execution* program as a loop.

```
range1^1 consumes $D$
repeat
    range1^1 generates a piece of $T$
    mult_reduce^1 consumes the piece of $T$ producing some of $R$
until $D$ is finished
```

This approach avoids excessive memory use by never generating T in its entirety. For comparison, we also express this computation with a serialized outer iterator.

```
 for (i=1;i<=#D;i++){
     T = range1(D[i]);
     R[i] = mult_reduce(T); }
```

In Table 1, we compare the execution of these two versions. In this example we have four processors and have set $D = [5, 2, 7, 3]$. For the serialized outer iterator code, if the size of T exceeds the number of processors, we must use multiple steps to complete the computation using virtual processors (see Fig. 1b).

Serialized Outer Iterator			
Step	Space	T	R
2	5	[1,2,3,4,5]	[120,]
1	2	[1,2]	[120,2,]
2	7	[1,2,3,4,5,6,7]	[120,2,5040,]
1	3	[1,2,3]	[120,2,5040,6]
Sum:6	Max:7		

Piecewise Execution			
Step	Space	T	R
1	4	[1,2,3,4	[ ]
1	4	5],[1,2],[1	[120,2,]
1	4	2,3,4,5	[120,2,]
1	4	6,7],[1,2	[120,2,5040,]
1	4	3]	[120,2,5040,6]
Sum:5	Max:4		

Table 1. Comparison of Approaches to Partial Serialization of Factorial Program

For comparison, executing the flattened program without any serialization on the four processor machine takes five steps (using virtual processors) and requires 17 memory locations. The serialized outer iterator approach does reduce the memory usage, but actually *increases* the number of steps due to undersized vectors. The piecewise execution maintains the minimal number of steps, and also significantly reduces the required memory. As the number of processors and the problem size increase, the cost of the undersized vectors increases and negates any gains made by serializing the outer iterators, while piecewise execution maintains effective memory use.

2.3 Requirements of piecewise execution

One of the key characteristics of the flattening technique is that it preserves the asymptotic work complexity of a computation, even for irregular, nested iterators. Therefore we require that piecewise execution also be work-efficient. We attempt to select a size for pieces that will keep the underlying parallel machine fully utilized. Oversized pieces exhaust memory resources, and undersized pieces fail to amortize the overhead of parallel execution or achieve a significant percentage of a parallel machine's peak performance. The proper piece size lies somewhere between $n_{1/2}$, which provides half the performance of the machine and $n_{\mathit{exceeds\ memory}}$ which cannot execute because of insufficient storage (See Fig. 5). Sethi [20] showed that determining whether a program can successfully execute without external memory using only k registers requires exponential time. Selecting an acceptable piece size is equivalently complex. The number of pieces, n and the piece size, p are inversely related by $n * p = M$. Since M, the total available memory for a parallel machine is fixed, selecting p determines n. Since we must select piece sizes with incomplete information, we require that the amount of serialization of piecewise execution be adjustable at runtime to support experimentation.

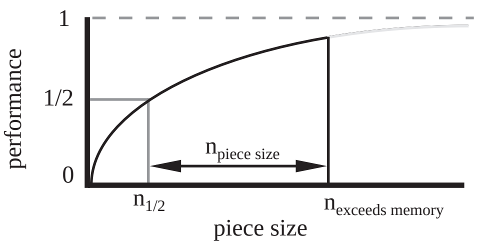

Fig. 5. Acceptable Values for the Size of Pieces

Also to maintain work efficiency, we avoid recomputing sequence values to facilitate piecewise execution. We do not require, but strive for efficient execution and minimal buffering of piecewise execution programs. In practice, there are several instances in which we cannot reach these goals. In Section 6, we examine two program configurations that may cause piecewise execution to fail. We also strive for an approach to piecewise execution that fosters chaining on a vector machine.

3 Interpreted piecewise execution

3.1 Piecewise primitive operations

Piecewise versions of the data-parallel primitive operations consume sequence inputs and generate sequence outputs in a succession of equal-sized pieces. To support fixed-memory execution of flattened programs, we must provide a piecewise version of every DPL operation. DPL consists of two types of operations: basic operations, including `range1` and `mult_reduce`, and data-parallel extensions of the basic operations, including `range1`^{1} and `mult_reduce`^{1}.

To implement piecewise versions of the basic operations, we straightforwardly consume pieces of input or generate pieces of output by initiating a series of calls to non-piecewise functions. For example, `piecewise_range(n,m,p)` enumerates of integers between n and m in pieces of size p by calling `range`, with the succession of values $(n, n+p-1), (n+p, n+2p-1), \ldots, (n+kp, m)$. Each consecutive invocation of `piecewise_range` generates the next consecutive piece of the overall sequence result.

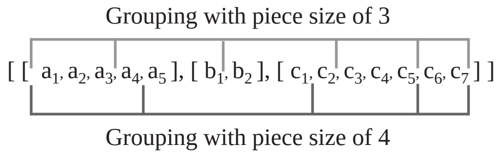
Grouping with piece size of 3
[[a1, a2, a3, a4, a5], [b1, b2], [c1, c2, c3, c4, c5, c6, c7]]
Grouping with piece size of 4

Fig. 6. Representative Groupings of Subsequences Across Pieces

Generating the piecewise versions of data-parallel operations requires a more sophisticated approach. There is, in general, no relationship between the piece size and the hierarchical boundaries of nested sequences (see Fig. 6). Piecewise versions of the data-parallel primitives must therefore maintain subsequence integrity across an arbitrary number of pieces. Consider the piecewise version of `range`^{1}`(`U`,`V`)` which generates, in pieces, the depth two sequence $[[U_1, \ldots, V_1], \ldots, [U_n, \ldots, V_n]]$. We implement the function `piecewise_range`^{1}`(`U`, `V`, `p`)` by making a series of calls to `range`^{1}, but we must carefully supply the proper input sequences.

We define r_k as the amount of space remaining within a piece after k subsequences have been generated. Whenever $V_k - U_k + 1 > r_{k-1}$, generating the k^{th} subsequence will require multiple iterations and computation of multiple starting and ending values of pieces within the subsequence. Conversely, whenever $V_k - U_k + 1 < r_{k-1}$ multiple subsequences will fit into a single piece, requiring a sequence of starting and ending values to generate enough of the result to fill that piece. Table 1 illustrates these conditions.

Both `range` and `range`^{1} are $generators$ because they can produce multiple output pieces from a single set of inputs. Additionally, their results are nested one level deeper than their inputs. Other operations, such as `mult_reduce` are $accumulators$ because they may consume multiple input pieces before producing an output. Results of accumulators are one level shallower than their inputs. Because of this structural correlation, generators that produce pieces for accumulators operate in step with each other. A third class of operations which have a one-to-one correspondence between consumption of input pieces and generation of output pieces, such as elementwise operations, are called $participants$. Table 2 provides a list and categorization of several representative data-parallel primitive operations.

Name	Action	Piecewise Behavior
arith-ops	basic arithmetic and logical operations	participant
substitute	replaces every sequence element with a supplied value	participant
distribute	replicate values to form a sequence	generator
range1	enumerate integers between 1 and a supplied value	generator
length	the number of elements in a sequence	accumulator
x_reduce	family of reduction operations ($+$, $*$, and, or, max, min)	accumulator
index	extract an element from a sequence	other
restrict	pack a sequence according to a mask	other

Table 2. Selected Nested Sequence Operations in Data Parallel Library

The operations `restrict` and `index` do not conform to any of the defined categories, so we describe their distinctive piecewise behavior individually. The `restrict` operation returns elements of an input sequence packed according to a boolean mask:

```
restrict([T,T,F,T,F],[1,2,3,4,5]) = [1,2,4]
```

Because the size of the result is determined by the value, and not the structure of the input (as with reduction), `restrict` may consume an arbitrary number of input pieces before generating a piece of result. This requires runtime size tests and eliminates the structure-based, lock-step execution of the factorial example.

Indexing operations also require special handling, because, unlike all other DPL operations, `index` consumes its input in a data-dependent order. Piecewise execution only works for operations that expect their inputs and generate their outputs in linear order. To satisfy arbitrary accesses, the entire source sequence must be available before indexing begins. As a result, `index` operations with piecewise-generated source sequences become *synchronization points*. All piecewise execution initiated prior to a synchronization point must complete before the `index` operation can begin. When its source sequence does not exceed the piece size, `index` does not require synchronization and can itself operate in a piecewise manner with respect to its indices.

Although this approach counteracts the effects of piecewise execution, Palmer, Prins and Westfold have developed another technique, *work-efficient indexing* [15], that prevents increasing the size of many source sequences during the flattening process. We expect that this will reduce the impact of index as a synchronization point on piecewise execution.

We implement piecewise versions of all primitive operations in C with explicit calls to DPL operations. These piecewise primitives comprise the Piecewise Data-Parallel Library (PDPL) which directly supports fixed-memory execution of flattened Proteus programs.

3.2 Retaining state between invocations of piecewise operations

Piecewise primitive operations behave like co-routines: many can be invoked simultaneously; only one executes at a time; they can suspend and resume execution; and they relinquish control to others after making some computational progress. To support this behavior, we introduce a new type to the Piecewise Data Parallel Library called an *engine*. Engines retain pertinent state information for piecewise operations so the operations can restart at the exact point where they previously suspended. Once restarted, the operation generates or consumes the next piece, updates the engine's state information to reflect the latest progress, and suspends (see Fig. 7).

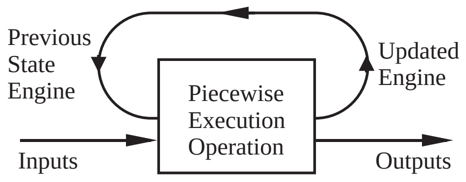

Fig. 7. Conceptual Model of an Engine

Each engine is associated with a single piecewise operation, and stores information specific to restarting that operation. Engines for generators retain the remaining portion of the operation's current piece of input which implicitly specifies the next output piece. Engines for accumulators retain the partially generated output piece and information on how to integrate the next input into the growing result. Participants directly produce an output piece from an input piece requiring no information from previous invocations and thus do not require engines.

Since engines contain all the inter-invocation information for piecewise operations, we allocate them in the heap. This allows us to achieve co-routine behavior without persistent activation records or altering the management of the stack.

3.3 Demand-driven piecewise interpretation

Pingali and Arvind [16, 17] use demand-driven interpretation to evaluate their stream-based language with infinite data structures. Unlike data-driven interpretation, this approach prevents non-termination and unbounded amounts of useless work. Although these issues do not impact Proteus, we use a modified form of demand-driven interpretation to support piecewise execution.

Ordinary demand-driven evaluation operates by demanding the output of the final node in the data-flow graph. Unable to comply without input, the final node propagates the demand to its parent nodes. Propagation continues in this manner until the demands reach to the top of the graph and can be satisfied by the inputs to the program, thus starting a cascade of node execution and generated data propagation.

Our approach differs from theirs in three significant ways. First, our data-flow graphs represent data-parallel programs, so aggregate values flow along the edges, not streams of scalar values. For efficient execution, our edges must always propagate piece-sized sequences so they can achieve a significant portion of a parallel machine's peak performance. Second, our data-flow graph contains generator nodes that can produce results without consuming any input, and it also contains accumulator nodes that can consume inputs without generating any output. To support this unusual behavior, we must handle demand propagation differently. Third, we localize the buffering of pending values to eliminate the need for unbounded storage along every edge.

The requirement to reduce memory usage of flattened nested data-parallel programs makes demand-driven interpretation attractive for piecewise execution. A generator cannot execute effectively in a data-driven style. An attempt to do so will either produce all its output pieces at once, possibly exceeding memory resources, or produce a single piece of output and relinquish control without a mechanism of regaining it. Demand-driven execution allows generators to produce single pieces in response to demands for single pieces, neither exceeding memory or abandoning results.

Our data-flow graphs have four basic types of nodes: generators, accumulators, participants and *copy nodes*. Copy nodes replicate values when a path in the dataflow graph splits. They are analogous to Pingali and Arvind's fork construct.

Node Type	Demand Action	Data Action
Generator	If possible, produce data else propagate demand upwards	Execute, and produce data
Accumulator	Propagate demand upwards	Execute, if possible produce data else propagate demand upwards
Participant	Propagate demand upwards	Execute and produce data
Copy	If data buffered for source of demand, send data, else mark source node as pending. If copy node is not waiting for a demand to be satisfied then propagate demand upwards	Send data to all pending nodes, buffer data for all other child nodes

Table 3. Demand Driven Actions for Piecewise Interpretation Nodes

Our demand-driven interpreter propagates data and demands according to the rules in Table 3. Each data produced and demand propagated is placed on an event queue. The event at the head is removed and executed, adding more events to the queue, when the queue is empty, the program is complete.

We generate a piecewise version of flattened programs from an abstract syntax tree representation of data-parallel operations. The structure of the piecewise program consists of three parts: a copy of the demand-driven interpreter, code to generate to the data-flow graph of the original program and modularized functions that encapsulate the operations of each node. This structure is analogous to the that of a table-driven parser, consisting of a general interpreter, a representation of the grammar, and action routines.

4 Piecewise Execution Loops

Interpreted piecewise execution successfully executes parallel programs in fixed memory. However, the generality of the approach incurs the overhead costs associated with interpretation. Additionally, the interpreted approach interferes with chaining sequence operations. One partial solution merges data-flow graph nodes that always execute consecutively, such as groups of participants. This provides some potential chaining of vector operations.

A more general solution compiles the data-flow graphs into *piecewise execution loops*. These loops extend between matching pairs of generators and accumulators as in Fig. 4. When generator/accumulator pairs are nested, the corresponding piecewise execution loops are also nested. An outer loop is necessary to restart the inner loop after it generates a single output piece.

```
repeat
        range1$^1$ consumes a piece of $D$
        repeat
                range1$^1$ generates a piece of $T$
                mult_reduce$^1$ consumes the piece of $T$ producing some of $R$
        until a full piece of $R$ has been produced or the piece of $D$ is finished
until all of $D$ has been consumed
```

In our early investigation into compiling piecewise execution loops, we identified several complex issues in statically producing code that emulates the behavior of demand driven execution. Identifying the generator/accumulator pairs that specify loop bounds is complicated by the possibility that multiple generators can match with a single accumulator and vice versa. Furthermore, pairs of generators/accumulators that exhibit the same piecewise structural behavior are *conformable* and should be placed in the same piecewise execution loop for best performance. Chatterjee's size inference [8] can be used to identify conforming operations. Restrict operations require the introduction of additional loops to provide the data-dependent number of input pieces necessary for restrict to generate an output piece. Finally, piecewise execution loops require a complex control-flow mechanisms to maintain small amounts of buffering, and perform the piecewise operations in the correct order. If we determine from our performance experiments that the overhead of interpreting piecewise execution programs is too costly, we will further investigate these compilation issues.

5 Experimental Results

The performance results shown in Fig. 8 illustrate piecewise execution's effective use of memory. The measured program computes multiple summations in parallel and requires 10 vectors of the maximum size to perform the computation. We impose a memory restriction of 1 Mword to highlight the differences between the direct and piecewise approaches. As a result, the direct calculation can only handle vector sizes up to 100,000 elements. For those vectors, the direct computation, as expected, yields better performance than piecewise execution. However, our results show that piecewise execution can perform the computation for dramatically larger problem sizes in the same amount of memory.

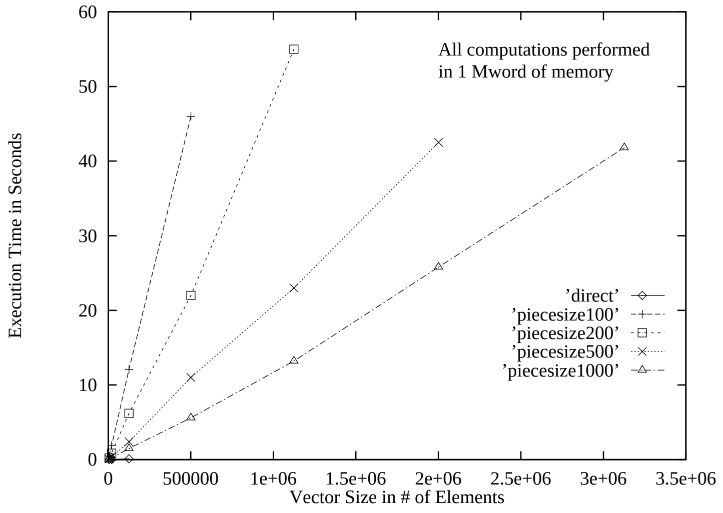

Fig. 8. Sequential Performance of Piecewise Execution

We also observe that the execution cost per element of the maximal vector size generally remains the same across the wide range of vector sizes. For small piece sizes the overhead associated with creating and maintaining an engine dominates the execution time. As the piece size increases, the effect of that cost diminishes and we start to see performance closer to that of the direct computation. We also ran this computation on the MasPar MP-1 and observed similar behavior, but the performance only got within a factor of 2 of the direct approach for piece sizes of 512K elements and larger.

6 Limitations of Piecewise Execution

Certain program configurations inherently preclude execution in a piecewise manner. Consider a program in which two generators consume the same piecewise generated input, but a data dependence between them prevents one from executing until the other completes. Because, in general, we cannot store the entire sequence, we must relax the execution constraints and allow the sequence to be recomputed. This increases the work, but allows the program to execute.

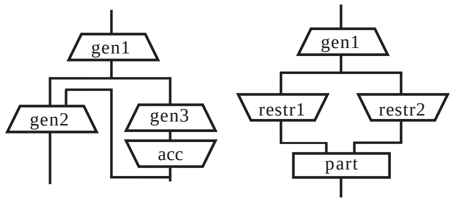

Fig. 9. Pathological Program Configurations

Another inherently difficult program configuration consists of multiple paths through the data-flow graph that contain `restrict` operations. The rate at which `restrict` consumes input and produces output is data-dependent. Therefore two `restrict` operations may consume the same input sequence at widely different rates. The slower of the *rate-divergent siblings* requires input buffering that can exhaust memory. We execute this configuration with normal piecewise execution, hoping the rates will be close enough to avoid memory problems. If they do exhaust memory, our only recourse is to again relax the “no recomputation” constraint, and execute the paths independently.

7 Discussion

7.1 Related work

Our key contribution in this paper is eliminating oversized temporaries from nested data-parallel programs, much previous work has been done applying this technique in other contexts. Reducing the memory requirements of a program by eliminating storage of temporary aggregate values is not a recent idea. In 1970, Abrams developed an interpreter system that partially compiled and sequentially executed APL programs [1]. The system postponed execution of certain operations until they could be optimized based on contextual information gathered during the postponement. These optimizations eliminated storing temporary aggregates, even those resulting from size increasing operations such as distributions and enumerations. Abrams accomplished this by processing a single element of the aggregate through an entire computation and yielding an element of the result before moving on to the next element. With this approach, he could evaluate expressions composed from a restricted set of APL operations using fixed storage equal to the larger of the expressions inputs and outputs.

In 1978, Guibas and Wyatt formalized and extended Abrams’ ideas to build a system that fully compiled APL programs [10]. Using data-flow analysis techniques, they generated code that statically did the equivalent of Abrams contextual postponement. The compiled code evaluated APL’s aggregate operations using fixed storage by streaming single elements of a flat aggregate through a complete computation.

Budd explored extending these ideas to the vector domain. Instead of single elements at a time, he proposed to stream a vector's worth of elements through a computation [7]. This approach not only evaluated APL expressions in space equal to the larger of the inputs and outputs, but could also make effective use of vector hardware.

Waters, generalizing the APL-based work to applicative series expressions for Common Lisp and Pascal, used data-flow analysis and program transformations to generate semantically equivalent imperative loop structures [21]. The transformations eliminated temporaries from series expressions and provided efficient single processor execution of functional programs using streaming. Although Waters speculated on extending his program transformations to handle nested series expressions, he did not implement it.

In 1993, Chatterjee compiled nested data-parallel programs to increase code granularity and relax lock-step synchrony so the programs could effectively execute on MIMD machines [8]. Although his compiler did not implement the fixed memory evaluation of Abrams, he was the first to apply temporary elimination in the context of nested data-parallel programs. His system used loop fusion to eliminate intermediate temporary storage from transformed NESL programs.

7.2 System status

We are currently building the piecewise execution system for Proteus on top of our existing execution system. We have implemented portions of PDPL and are currently working on implementing the rest. We have written the demand-driven piecewise interpreter with the modified demand driven execution and run it on small programs with large datasets. We have not yet automated the generation of piecewise programs or the compilation of piecewise execution loops. Our current research focuses on piecewise execution of user-defined functions, and our current implementation effort is aimed at reducing the memory management costs associated with creating and using engines.

7.3 Conclusions

The major drawback to the technique of flattening nested data parallelism is that it extracts so much parallelism that it often generates programs that cannot execute within the memory limitations of parallel machines. In this paper we presented piecewise execution, an approach which provides parametric runtime serialization of flattened parallel programs to overcome this obstacle. Our preliminary results confirm that piecewise execution can significantly reduce the memory requirements of a flattened, nested data-parallel program. Future results will reveal whether interpreted piecewise execution will provide sufficient performance or if we must further develop compilation techniques and generate piecewise execution loops. Regardless of our ultimate approach to implementing piecewise execution, we have demonstrated its applicability and usefulness.

References

1. P. Abrams. *An APL Machine*. PhD thesis, Stanford University, 1970.
2. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–41, Aug. 1978.
3. G. Blelloch and G. Narlikar. A comparison of two n-body algorithms. In *Proceedings of DIMACS Parallel Implementation Challenge Workshop III*, Oct. 1994.
4. G. Blelloch and G. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2), Feb. 1990.
5. G. E. Blelloch. Nesl: A nested data-parallel language. Technical Report CMU-CS-92-129, Carnegie Mellon University, 1992.
6. G. E. Blelloch, S. Chatterjee, J. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. Cvl: a c vector library manual, version 2. Technical Report CMU-CS-93-114, Carnegie Mellon University, 1993.
7. T. Budd. *An APL Compiler*. Springer-Verlag, 1988.
8. S. Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM Trans. Prog. Lang. Syst.*, 15(3):400–462, July 1993.
9. H. P. F. Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, 1993.
10. L. J. Guibas and D. K. Wyatt. Compilation and delayed evaluation in APL. In *Conf. Record of the Fifth Annual ACM Symp. on Princ. of Prog. Lang. (Tucson, Arizona)*, pages 1–8. ACM, Jan. 1978.
11. P. Hatcher and M. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.
12. K. Iverson. *A Programming Language*. Wiley, 1962.
13. G. Levin and L. Nyland. An introduction to Proteus, version 0.9. Technical report, University of North Carolina at Chapel Hill, Aug. 1993.
14. D. W. Palmer. Dpl: Data-parallel library manual. Technical Report UNC-CS-93-064, University of North Carolina at Chapel Hill, Nov. 1993.
15. D. W. Palmer, J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Proc. Fifth Symp. on the Frontiers of Massively Parallel Processing (Frontiers 95)*. IEEE., 1995.
16. K. Pingali and Arvind. Efficient demand-driven evaluation. Part 1. *ACM Trans. Prog. Lang. Syst.*, 7(2):311–33, Apr. 1985.
17. K. Pingali and Arvind. Efficient demand-driven evaluation. Part 2. *ACM Trans. Prog. Lang. Syst.*, 8(1):109–39, Jan. 1986.
18. J. F. Prins and D. W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proc. 4th PPOPP. (San Diego, CA, 19–22 May 1993)*. ACM., 1993. Published in SIGPLAN Notices, 28(7):119–28.
19. J. Schwartz. Set theory as a language for program specification and programming. Technical report, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, 1970.
20. R. Sethi. Complete register allocation problems. *SIAM Journal of Computing*, 4(3), 1975.
21. R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. Prog. Lang. Syst.*, 13(1):52–98, Jan. 1991.

This article was processed using the LaTeX macro package with LLNCS style