Piecewise Execution of Nested Parallel Programs - A Thread-Based Approach

W. Pfannenstiel

Technische Universität Berlin
wolfp@cs.tu-berlin.de

Abstract. In this paper, a multithreaded implementation technique for piecewise execution of memory-intense nested data parallel programs is presented. The execution model and some experimental results are described.

1 Introduction

The flattening transformation introduced by Blelloch & Sabot allows us to implement nested data parallelism while exploiting all the parallelism contained in nested programs [2]. The resulting programs usually have a high degree of excess parallelism, which can lead to increased memory requirements. E.g., in many programs, operations known as generators produce large vectors, which are reduced to smaller vectors (or scalars) by accumulators almost immediately afterwards. High memory consumption is one of the most serious problems of nested data parallel languages. Palmer, Prins, Chatterjee & Faith introduced an implementation technique known as *Piecewise Execution* [4]. Here, the vector operations work on vector pieces of constant size only. In this way, low memory bounds for a certain class of programs can be guaranteed. One algorithm in this class is, e.g. the calculation of the maximum force between any two particles in N-body computations [1].

In this paper, we propose a multi-threaded implementation technique of piecewise execution. Constant memory usage can be guaranteed for a certain class of programs. Furthermore, the processor cache is utilized well.

2 Piecewise Execution

The NESL program presented in Figure 1 is a typical example of a matching generator/accumulator pair. The operation `[s:e]` enumerates all integers from `s` to `e`, `plus_scan` calculates all prefix sums, `{x * x : x in b}` denotes the elementwise parallel squaring of `b`, and `sum` sums up all values of a vector in parallel. Typically, there is a lot of excess parallelism in such expressions that must be partly sequentialized to match the size of the parallel machine. However, if the generator executes in one go, it produces a vector whose size is proportional to the degree of parallelism. In many cases, then, memory consumption is so high that the program cannot execute.

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 445–448, 1999.

```
function SumSqScan (s, e) =
  let
    a = [s : e];
    b = plus_scan (a);
    c = {x * x : x in b}
  in
  sum (c);
```

Figure 1: Example program

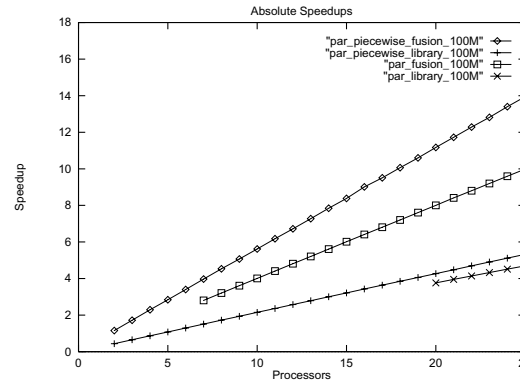

Figure 2: Speedups

In piecewise execution, each operation receives only a piece, of constant size, of its arguments at a time. The operation consumes the piece to produce (a part of) its result. If the piece of input is completely consumed, the next piece of input is requested. Once a full piece of output is produced, it is passed down as an input piece to the next vector operation. Thus, large intermediate vectors never exist in their full length. However, some means of control are needed to keep track of the computation. In [4], an interpreter is employed to manage control flow. Unfortunately, interpretation involves a significant overhead.

3 Execution Model and Implementation

We propose an execution model for piecewise execution using multiple threads. Threads in our model cannot migrate to remote processors. At the start of a thread, arguments can be supplied as in a function call. A thread can suspend and return control to the thread (or function) from which it was called or restarted using the operation `switch_to_parent`. Each thread has an identifier, which is supplied to `restart_thread` to reactivate it from anywhere in the program.

3.1 Consumer/Producer Model

A piecewise operation consumes one input piece and generates one output piece at a time. In general, a generator produces numerous output pieces from one input piece, whereas an accumulator consumes many pieces of input before completing a piece of output. Elementwise vector operations produce one piece of output from one piece of input. In the program DAG, the bottom node (realized as p threads on p processors cooperating in SPMD style) calculates the overall result. It requests a piece of input from the node(s) on which it depends. The demand is propagated to the top thread, which already has its input. Whenever a thread completes a piece of output, it suspends and control switches to the consumer node below. If a thread runs out of input before a full piece of output

has been built, it restarts (one of) its producer node(s). In this way, control moves up and down in the DAG until the last node has produced the last piece of the overall result.

Only certain parts of a flattened program should run in a piecewise fashion. Thus, it must be possible to switch seamlessly between ordinary and piecewise execution mode. There are two special thread classes `pw_in` and `pw_out`, which provide an entry/exit protocol for piecewise program fragments. A `pw_in` thread consumes an ordinary vector at once, which is then supplied in pieces to the underlying threads of the piecewise program fragment. Thus, `pw_in` is always the topmost producer of data. However, it does not really produce data. It copies its argument vector(s) to the output in pieces. It never restarts any thread, because its input is supplied at once at call time. At the other end of the piecewise program fragment, `pw_out` is the last consumer of data. It collects all data pieces from above until the last piece is finished. Then, the complete result is passed on to the remaining part of the program, which works in an ordinary manner. An instance of `pw_out` never calls `switch_to_parent` but uses `return` after the complete result has been assembled. Often, `pw_in` and `pw_out` are not required, if – like in the example – scalars are consumed or produced. They are atomic values requiring constant space, making it impossible to supply them in pieces.

In our execution model the computation is triggered by the last thread, and the demand for data is propagated upwards in the program DAG. This resembles a demand-driven dataflow-like execution. However, to have normal execution order in the whole program, the threads in the piecewise program part are spawned in the same order as the other operations. First the producer is called, then the consumer. When a thread is spawned, it returns a *closure* and suspends. A closure consists of a thread handle and the address of a buffer for its output. The consumer receives the closure of its producer as an argument to be able to restart it if it needs data.

The threads are self-scheduled, i.e control switches explicitly among threads. For every piecewise operation, there is exactly one thread per processor and the structure of calls, suspends and restarts is the same on every processor. The threads constituting a piecewise operation run in sync with one another and use barriers to synchronize.

4 Experiments and Benchmarks

To test the feasibility of our approach, we transformed the example program `SumSqScan` from Figure 1 manually and implemented it on the Cray T3E. Keller & Chakravarty have proposed a new intermediate language DKL, whose main feature is the separation of local computations from communication of flattened parallel programs [3]. Local computations can be optimized by a set of transformations, the most important optimization being the fusion of consecutive loops. For our experiments, we tried to combine the proposed optimizations with piecewise execution. Thus, we implemented four combinations of features: A plain library version, a fused version in which parallel operations are fused

as much as possible and the piecewise variants of both. The programs are written in C in SPMD style using the StackThreads library [5]. Communication and synchronization among processors is realized using the native Cray `shmem` communication library. We ran the four versions of the program for different parameter ranges and calculated the absolute speedups. The speedups are shown in Figure 2. The piece size was set to the respective optimum of the piecewise version, which corresponds to the size of the processor cache. Surprisingly, the combination of fusion and piecewise techniques yields the fastest results. Fusion leads to a significant improvement, but employing piecewise execution leads to shorter runtimes in this example, too. The optimized use of the processor cache would appear to compensate the overhead incurred by the multithreading implementation. Furthermore, the improvement of fusion and piecewise execution seem to mix well. Both techniques reduce memory requirements in an orthogonal way. The pure library and fusion versions cannot execute in some situations because of memory overflow. The input size of the piecewise versions is limited only by the maximum integer value.

5 Conclusion and Future Work

We have presented a new implementation technique for piecewise execution based on cost-effective multithreading. We have shown that piecewise execution does not necessarily mean increasing runtime. On the contrary, the combination of program fusion and piecewise execution resulted in the best overall performance for a typical example. Piecewise execution allows us to execute a large class of programs with large input sizes that could not normally run owing to an insufficient amount of memory.

We intend to develop transformation rules that automatically find and transform program fragments suitable for piecewise execution for use in a compiler.

References

[1] J. E. Barnes and P. Hut. Error analysis of a tree code. *The Astrophysical Journal Supplement Series*, (70):389–417, July 1989.

[2] G. E. Blelloch and G. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, 1990.

[3] G. Keller and M. M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In *Proceedings of the Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99)*. IEEE CS, 1999.

[4] Daniel Palmer, Jan Prins, Siddhartha Chatterjee, and Rik Faith. Piecewise execution of nested data-parallel programs. In *Languages and compilers for parallel computing : 8th International Workshop*. Springer-Verlag, 1996.

[5] K. Taura and A. Yonezawa. Fine-grain multithreading with minimal compiler support - a cost effective approach to implementing efficient multithreading languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1997.