Streaming Extensions for Object-Oriented Languages

Workshop on Streaming Systems: From Web and Enterprise to Multicore
41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)
November 8, 2008, Lake Como, ITALY

Frank Otto
University of Karlsruhe
76131 Karlsruhe, Germany
otto@ipd.uka.de

Victor Pankratius
University of Karlsruhe
76131 Karlsruhe, Germany
pankratius@ipd.uka.de

Walter F. Tichy
University of Karlsruhe
76131 Karlsruhe, Germany
tichy@ipd.uka.de

Abstract

Stream languages provide constructs to express different types of parallelism, such as pipeline parallelism, in a simple way. In the past, these languages were mainly used for the development of signal processing or graphics applications. We argue that the integration of stream programming concepts into universal object-oriented languages has great potential to simplify general-purpose parallel programming on multicore architectures.

1. Introduction

As clock rates are stagnating and multicore chips become mainstream, parallel programming becomes a concern for many developers. Throughout the years, threading has been dominating the world of parallel programming. However, it is well-known that this way of programming is error-prone and difficult on a large scale. Motivated by these developments, a new search for parallel programming models has begun.

In stream programs, the entire code is written in a sequential pipe-and-filter style that is amenable to automatic parallelization by a compiler. Existing approaches like StreamIt [1], [5] have demonstrated the applicability of this model in specialized domains such as signal processing or graphics. Several types of parallelism, such as task, data, pipeline, or nested parallelism, were shown to be easily exploitable.

Despite the narrow focus, streaming concepts are useful for general-purpose programming as well. For example, a concise expression of pipeline processing would reduce code size, limit sources of error, improve understanding, and hide some details of parallelization. Therefore, it seems promising to combine the best of both worlds: object-oriented languages offer universality and address a broad community of developers, while stream languages simplify the usage of various types of parallelism.

2. Streaming Extensions for Object-Oriented Languages

To express parallelism in object-oriented languages, developers are typically required to think on a low abstraction level where threads are explicitly created and destroyed. Very often, this is the only level where parallelization is considered. Another problem is that even experienced programmers may produce incorrect parallel code, forget about nondeterministic execution, and make wrong assumptions about thread interleavings.

Our case studies on the parallelization of real-world applications [3], [4] show that it is advantageous to exploit parallelism on several abstraction levels. Figure 1 illustrates the architecture of an application performing biological data analysis. On the highest abstraction level, there is a pipeline structure that is subsequently refined. Inside a pipeline stage, processing may be controlled by master-worker or producer-consumer patterns [2]. Continuing the refinement, split-join mechanisms are used inside lower-level modules to carry out data-parallel computations.

Many complex applications use such nested parallelism, but require the flexibility of object orientation. Therefore, it seems reasonable to integrate streaming concepts into object-oriented programming. The streaming model simplifies the expression of pipeline parallelism on high abstraction levels.

2.1. Libraries versus Language Extensions

There are two basic approaches to extending object-oriented languages by streaming constructs: (1) libraries and (2) native language extensions (i.e. compiler extensions).

(1) Libraries can be implemented in existing languages. Compared to a compiler extension, creating a library is a short-term solution and requires less effort. However, certain language limitations might prevent composability in the

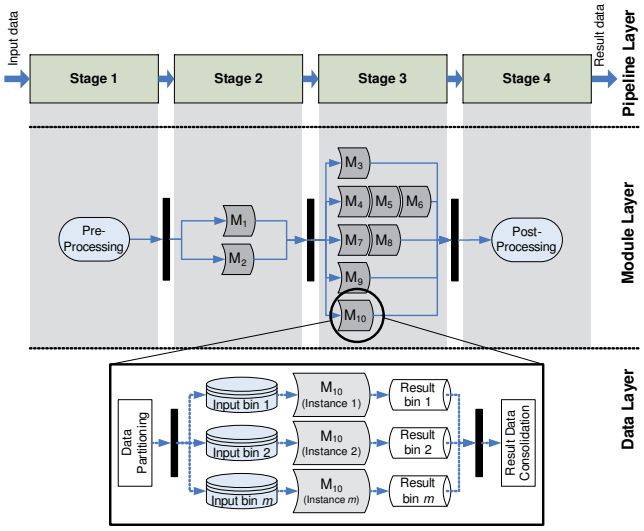

Figure 1. Conceptual architecture of a parallel application for biological data analysis [4].

parallel case. Debugging is difficult as well, because the compiler has no knowledge about the semantics of streaming constructs.

(2) Language extensions are more complex to implement, but also more promising in the long run. In this approach, streaming constructs can be expressed using an intuitive syntax, and the compiler is aware of streaming semantics. A stream computation is basically expressed by a special fragment (as simple as a Unix pipe) embedded in a general-purpose program. Compared to explicit threading, we consider this approach to be much more intuitive.

Beyond that, the stream semantics can be used by the compiler to automatically exploit data parallelism. For example, consider a certain stage that is replicable to other available cores. In a situation with heavy load, this stage would automatically be cloned and executed in parallel.

As a **proof of concept**, we started working on language extensions for Java, which are sketched next. Similar to filters in stream languages, our Java extension provides a *task* construct, which is a special kind of method that can have a dedicated input and output. A task `t` with input type `X` and output type `Y` is declared as `task[X => Y] t()`; if it has no input or output, the corresponding type is `void`. A `work` block within a task's body implements a loop to process the elements of the input stream. The body of that loop is executed until no more input elements are available, or (if the task has no input) as long as a conditional expression is true. Finally, a `push` statement puts an element to the output stream.

For illustration, consider a file compression application [3], which hypothetically provides the classes `Block` and `File`. The algorithm divides an input `File` into fixed-sized `Block`s, compresses them, and stores the compressed `Block`s in the original order in an output `File`. Using a pipeline, the stages are expressed as follows:

```
public task[void => Block] read(File f) {
  Iterator i = f.getBlocks();
  work(i.hasNext()) { push (Block) i.next(); }
}

public task[Block => Block] compress() {
  work(Block b) { push b.compressBlock(); }
}

public task[Block => void] write(File f) {
  work(Block b) { f.add(b); }
}
```

The next code fragment creates the pipeline in Java, handling pipeline parallelism in a transparent way:

```
read(inFile) => compress() => write(outFile);
```

The "`=>`" operator defines connectors between stages. Similar operators were implemented for splitting and joining streams.

Even for such a small example, the potential code savings become evident when compared to explicit threading. Furthermore, no explicit locking or synchronization constructs were needed.

3. Conclusion

In the multicore era, the scalability of parallel programs is essential and depends on the exploitation of parallelism on all fronts. The integration of streaming concepts into object-oriented languages may improve the state of the art in several ways: pipeline parallelism and nested parallelism are easier to express, parallel programs are easier to understand and maintain, and the potential for errors is reduced. However, further research is required to explore all details and trade-offs of our approach; this is ongoing work in our group.

Acknowledgment. We thank the University of Karlsruhe and the Excellence Initiative for their support.

References

[1] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs, *Proc. ASPLOS-XII*, pp. 151–162. ACM, 2006.

[2] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Boston, 2005.

[3] V. Pankratius, A. Jannesari, and W. F. Tichy. Parallelizing bzip2. A case study in multicore software engineering. Accepted September 2008 for IEEE Software.

[4] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *Proc. IWMSE '08*, pp. 53–60. ACM, 2008.

[5] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In R. N. Horspool, editor, *CC*, LNCS volume 2304, pp. 179–196. Springer, 2002.