High-Level Multicore Programming with XJava

Frank Otto, Victor Pankratius, and Walter F. Tichy
University of Karlsruhe
76131 Karlsruhe, Germany
{otto, pankratius, tichy}@ipd.uka.de

Abstract

Multicore chips are becoming mainstream, but programming them is difficult because the prevalent thread-based programming model is error-prone and does not scale well. To address this problem, we designed XJava, an extension of Java that permits the direct expression of producer/consumer, pipeline, master/slave, and data parallelism. The central concept of the extension is the task, a parallel activity similar to a filter in Unix. Tasks can be combined with new operators to create arbitrary nestings of parallel activities.

Preliminary experience with XJava and its compiler suggests that the extensions lead to code savings and reduce the potential for synchronization defects, while preserving the advantages of object-orientation and type-safety. The proposed extensions provide intuitive "what-you-see-is-what-you-get" parallelism. They also enable other software tools, such as auto-tuning and accurate static analysis for race detection.

1. Introduction

Dual- and quad-core CPUs have become standard in today's desktop computers; a doubling of cores with every chip generation is expected. The shift to parallelism has amplified interest in parallel programming; parallelism is no longer confined to scientific computing, database systems, or instruction-level compiler optimizations. Many applications will be parallel in the future, and software research needs to address the issues regarding general-purpose parallel programming.

Writing parallel programs is difficult, because programming abstractions are typically limited to threads with explicit synchronization. As a result, many defects arise due to atomicity violations, order violations, or deadlocks [5]. We think that parallel programming can be significantly simplified. In our view, the currently available concepts and languages deal with parallelism at an abstraction level that is too low. Promising concepts exist, but are highly specialized and inaccessible in a broader context. For example, stream languages such as StreamIt [11] provide intuitive constructs for parallelism, but were designed for signal processing and graphics. We need to find ways to extend the key concepts and make them accessible in a fully-featured, general-purpose programming language.

In this paper, we present XJava, a programming language that extends Java with parallelism in a novel way. We unify key concepts from stream languages [3], parallel design patterns [6], and object-orientation. Our language design is motivated by observations made in previous case studies [8] and by the need to express parallelism in a simple and intuitive way. The central language constructs are the *task* and a set of combination operators. A task is basically an extended method; it runs in its own thread and provides a stream-oriented interface. It may be used with combination operators for expressing consumer/producer or master/slave configurations, pipelines, data parallelism, and other parallel patterns. Preliminary results indicate that this approach provides productivity improvements over threaded Java, while delivering satisfactory speedups.

2. Related Work

Stream-oriented languages are a domain-specific approach to parallel programming. A stream program consists of interconnected filters. The input is a theoretically infinite stream of data elements "flowing" through the filters. A survey of stream languages can be found in [10]. Languages such as StreamIt have been demonstrated to express data, task, and pipeline parallelism for programs in the domain of signal processing and graphics [3]. However, these languages provide rather simple data structures and do not offer the flexibility of object-orientation.

X10 [2] is an experimental language for parallel and distributed programming with focus on cluster computing. Fortress [1] provides implicit parallelism and supports the concept of transactions. However, both are are optimized for numeric programs. Streams and master/slave parallelism are not explicitly addressed.

The alternative to language extensions are libraries. The `java.util.concurrent` package provides constructs for multithreading in Java [4]. Intel's Threading Building Blocks (TBB) is a library for concurrent C++ programming [9]. However, library solutions are not as intuitive and succinct as well-designed language extensions. While libraries require less effort to implement than compilers, composability is often limited. Libraries primarily encapsulate functionality without providing semantic information to the compiler. Appropriate language constructs, on the other hand, provide additional semantic information which enables optimizations, better tools, easier debugging, improved understanding, code savings, and higher productivity.

3. Case Studies on Parallel Programming and Parallel Design Patterns

We recently carried out case studies on the parallelization of real-world applications [8]. These case studies were intended for learning about parallelization strategies and adequate designs of parallel applications. We observed that parallelism occurred on several levels of granularity. For example, Figure 1 illustrates the architecture of a parallel application for biological data analysis. On the top level, the architecture is a pipeline. At the next level of granularity, stages 2 and 3 exhibit task parallelism, with some of the tasks being pipelines. At the next level down, data parallelism is employed. Split-join mechanisms are used for task parallelism and low-level data parallelism. Implementing nested parallel architectures such as this one with threads is painful; the resulting code is hard to understand, tune, maintain, and extend.

In our case studies we also found that many features of object-oriented languages are useful when parallelizing sequential code, so we decided to add the required concepts to a general-purpose object-oriented language.

4. Unifying Object-Orientation, Streams, and Patterns in XJava

In this section, we describe XJava's language extensions and implementation. The central construct is the *task*; tasks can be combined to *parallel statements*.

4.1. Tasks

Syntactically, tasks are special methods. Tasks can be declared in interfaces or classes and inherit or override other tasks. Their properties can be specified by common flags

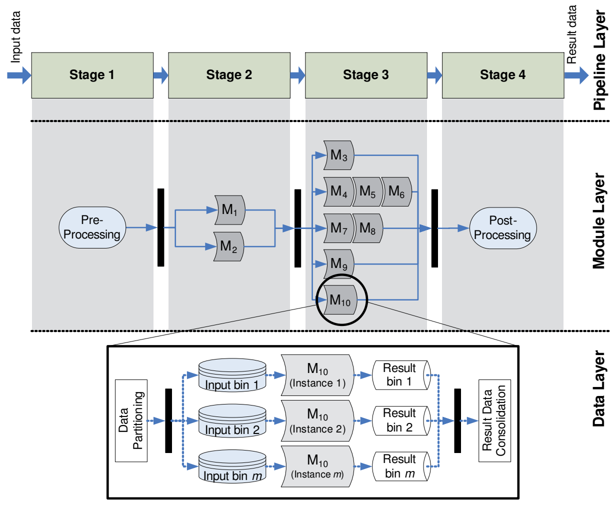

Figure 1. Conceptual architecture of a parallel application for biological data analysis [8].

such as `protected`, `static`, `abstract`, or `final`. They may throw or catch exceptions. A task runs as a separate thread and has an input and an output port. It receives a stream of data at its input port and generates a stream of data at its output port. Both ports are typed. For example,

```
public X => Y foo() { /* ... */ }
```

declares a public task `foo` with input type `X` and output type `Y`. That is, input data has to be of type `X` or its subtypes; output data will be of type `Y` or its subtypes. These types may be `void` if there is no input or output. Depending on the structure of the task body, we distinguish periodic and non-periodic tasks, as described next.

Periodic tasks define exactly one `work` block inside their bodies. A `work` block is repeatedly executed as long as there is input available at the task's input port. At the start of each iteration, the next object received at the input port is assigned to a local variable declared after the keyword `work`. A `push` statement puts an object on the output stream. Thus, periodic tasks are similar to Unix filters, except that their inputs and outputs can be any data type. Filters whose input type is `void` also repeat, but their stopping condition is given by a boolean expression after the `work` keyword. As soon as a periodic task receives an end-of-stream token or its boolean expression returns false, it pushes an end-of-stream token into its output stream and the repetition of the `work` block terminates. Additional code may appear before or after the `work` block; it is executed when the task starts or finishes, respectively.

For illustration, consider a file compression application. The algorithm divides an input `File` into fixed-sized `Block`s, compresses each of them, and stores the compressed `Blocks` in the original order in an output `File`. This scheme is used in applications such as BZip2. The tasks for reading, compressing and writing could be expressed as follows:

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

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

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

Non-periodic tasks do not define a work block and are executed only once. They may contain parallel statements, i.e. combinations of task calls that spawn nested parallelism (cf. next Section). The compiler ensures that input and output types of the nested parallel statement correspond to those input and output types of the surrounding non-periodic task.

4.2. Parallel Statements

Tasks can be combined by new operators to introduce parallelism. Such a combination is called a parallel statement. There are two major types of operators, with several variants.

The "`=>`" operator creates a *pipe statement*. It connects tasks via their input/output ports and builds pipelines. For the file compression example, the statement

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

creates a pipeline that automatically works in parallel. A simple producer/consumer configuration is a pipeline with two stages.

The "`|||`" operator creates a *concurrent statement*. The operands are tasks that will be executed in parallel. Suppose we want to compress two files concurrently. Then we would declare a non-periodic task

```
compress(File in, File out) {
  read(in) => compress() => write(out);
}
```

encapsulating the pipeline above. The statement

```
compress(f1, f1out) ||| compress(f2, f2out);
```

would compress the files `f1` and `f2` in parallel and store the results in `f1out` an `f2out`, respectively. There is an implicit barrier at the end of each parallel statement; the next statement is only executed after the tasks called in the parallel statement are finished.

Finally, we illustrate how to build a master/worker configuration, using nested parallelism. First, we need the workers. They are declared as a concurrent statement wrapped inside a non-periodic task, say task `b()`. The concurrent statement provides a static number of workers; a shorthand allows the number of workers to be determined dynamically. Next, the expression `a() => b()`, where `a()` is the master task, feeds the workers in a round-robin fashion with data objects. Variants of this operator are "`=>?`" for indeterministic distribution on a first-come-first-serve basis, and "`=>*`" for broadcasting objects to all tasks. These operators simplify the implementation of several common patterns described in [6].

4.3. Language Implementation

We built a preprocessor that translates the XJava extensions to native Java, using the Polyglot compiler framework [7]. In addition, we built a prototype runtime system consisting of a scheduler and other classes for runtime management. The scheduler controls the number of running threads and could enable future performance optimizations at runtime.

5. Results

We tested the XJava language constructs in several contexts to study their fitness for general parallel applications, not just the streaming domain. First, we tried out simple examples such as text transformations, sorting algorithms, matrix multiplications and mandelbrot sets. We created sequential and parallel versions to compare performance. Speedups ranged between 2.0 and 3.5 on a quad-core machine.

For a few programs, we wrote versions employing explicit threading. Up to 40% of the lines of code could be saved using XJava.

Furthermore, we implemented the skeleton of the parallelized application introduced earlier (cf. Figure 1) using our language extensions. The following sketch of task declarations illustrates the design:

```
void => X stage1() {...}
X => Y stage2() { m1() ||| m2(); }
Y => Z stage3() { m3() ||| m456() ||| m78() |||
                  m9() ||| m10(); }
Z => void stage4() {...}
...
Y => Z m456() { m4() => m5() => m6(); }
Y => Z m78() { m7() => m8(); }
...
X => Y m1() {...}
...
Y => Z m10() { m101() ||| m102() ||| m103(); }
```

The types X, Y, and Z signify the classes of objects generated by each stage. Finally, the statement

```
stage1() =>* stage2() =>* stage3() => stage4();
```

creates the top-level pipeline with nested parallelism as shown in the figure.

6. Improvements for Software Engineering

Based on our application studies, we believe that the proposed language extensions will make software engineering of general-purpose parallel applications easier. The power of object-oriented paradigms does not need to be sacrificed. The proposed extensions provide a clear, "what-you-see-is-what-you-get" syntax for parallelism. An XJava program is more likely to behave as expected by developers than a threaded form. As many users already know how Unix pipes work, we built upon this metaphor to express pipeline parallelism in ordinary code.

XJava's abstraction mechanisms allow programmers to "think in tasks". Wherever possible, XJava hides the confusing details of thread creation and destruction, locks, signals, or buffers. Thus, it is less likely that programmers forget something or make wrong assumptions about the program's behavior. XJava's operators are composable – a property that is largely neglected, but which is important for the engineering of large parallel applications.

Debugging becomes easier since the compiler knows the semantics of native language constructs and could provide more precise analyses. In addition, our case studies show that compared to programs with explicit threading, significant code savings can be expected.

7. Conclusion

XJava simplifies the expression of parallelism in general-purpose parallel applications. The main concepts of its programming model are the task abstraction and composition operators. Tasks can be composed to express different types of parallelism on different levels of granularity. Several test programs showed promising results: the code was simpler, shorter, required no explicit synchronization, and provided less room for bugs.

Further research is needed to asses which other extensions are useful. For this purpose, future case studies are planned. Static and dynamic analysis could take advantage of the semantics of the parallel constructs, in order to provide precise happens-before and may-happen-in-parallel information. Even with the constructs proposed here, it is still possible to produce data races and deadlocks. With precise analysis, these problems could be localized accurately. The integration of auto-tuning strategies to improve performance on different platforms is another area that might benefit from the semantic information contained in tasks and the combination operators.

Acknowledgments. We thank the University of Karlsruhe and the Excellence Initiative for their support. We also thank David Meder for assistance with the experiments.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification 1.0, April 2008.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. OOPSLA '05*, pages 519–538. ACM, 2005.

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

[4] D. Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58(3):293–309, 2005.

[5] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS XIII*, pages 329–339. ACM, 2008.

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

[7] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In G. Hedin, editor, *CC*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2003.

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

[9] J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Inc, Sebastopol, 2007.

[10] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[11] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In R. N. Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer, 2002.