OpenMP for Java

Petr Bělohlávek

JUDr. Antonín Steinhauser

Motivation

Due to its backward compatibility, multi-threaded programming in Java is often performed ineffectively and repels inexperienced programmers from development of parallel applications.

The directive based programming appears to be a solution for this problem. However, the current OpenMP implementations for Java lack both flexibility and portability. Furthermore, the current solutions often do not support Java 8 or produce code that requires an installed runtime library.

This project - omp4j - is an OpenMP-like preprocessor that is portable and the processed code has no runtime dependencies.

Features & Methodology

The implemented preprocessor supports Java language standards 6, 7 and 8 and it may be used on all commonly used JDKs. Additionally, the output code does not require any runtime dependencies.

The preprocessor may be employed either as a source-to-source preprocessor or as a source-to-bytecode compiler.

The combination of the syntax and bytecode analyses is employed by the preprocessor in order to parallelize the source code. By using the advanced code analysis, the preprocessor makes only local changes and keeps the source code modified as less as possible.

Performance Evaluation

The performance evaluation, which discusses various aspects of the implemented preprocessor, is presented and the comparisons to the related C++ and Java solutions are elaborated.

In comparison with the similar Java projects, significantly better scalability of omp4j is proven at significance level $\alpha = 0.01$ for some of the benchmarks. Figure 3 demonstrates the measured speedup in context of some related projects.

Furthermore, the measured speedup is modeled by using regression. Figure 4 provides visualization of an estimated linear model.

Conclusion

The implemented preprocessor is highly portable and extendable. It supports all modern Java standards and may be easily employed on all commonly used JVMs.

Furthermore, the differences between the input and output code are minimized. The implemented OpenMP grammar allows the user to employ various permutations of directive attributes.

In addition, the parallelized output is comparable with gcc compiled binaries. The preprocessor provides significantly higher scalability than related Java project JOMP.

Figure 1
OpenMP specification employs the Fork-join model. This figure demonstrates the thread creation and parallel task execution. As it is demonstrated, each set of tasks may be executed using different number of threads. The runtime model of omp4j ensures efficient thread recycling.

Figure 2
The preprocessor may be used either as a source-to-source preprocessor or as a source-to-bytecode compiler. Furthermore, it accepts all javac options, hence it may be used within any modern IDE.

Figure 3
The plots demonstrate the provided speedup when using different OpenMP implementations. One thousand (the left chart) and twenty thousand (the right chart) Fibonacci numbers are computed independently on 48 CPU NUMA computer. According to the plots, the scalability of translated code is sufficient as the non-linearity is probably caused by Amdahl’s law.

Figure 4
The provided speedup is modeled by a linear model in form $CORESi = \beta_0 + \beta_1 CORESi + \beta_2 CORESi + \epsilon$, which fits well.